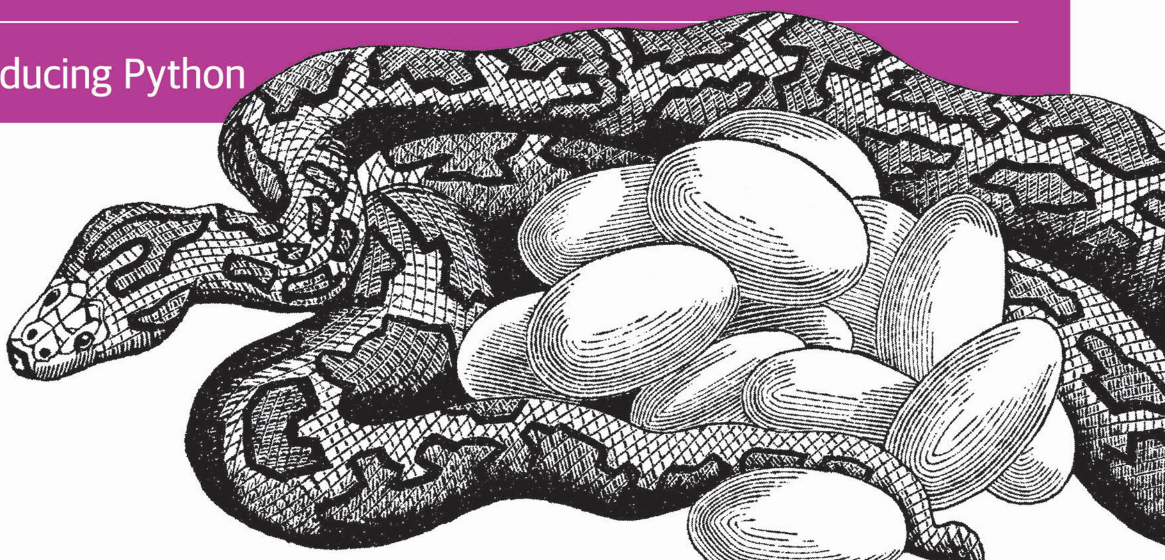
O'REILLY

图灵程序设计丛书

Python 语言及其应用

Introducing Python



[美] Bill Lubanovic 著

丁嘉瑞 梁杰 禹常隆 译

中国工信出版集团 人民邮电出版社 POSTS & TELECOM PRESS

数字版权声明

译者介绍



丁嘉瑞

耶鲁大学计算机硕士，曾就读于北京理工大学。主要研究方向为机器学习，曾从事iOS开发。个人博客：Jiarui-blog.com。



梁杰

前端工程师，Swift爱好者，Python爱好者。热爱开源和分享，目前正在维护SwiftGG翻译组、Swift开发者周刊。@Swift大会发起者，希望通过自己的行为来推动技术发展。



禹常隆

2015年本科毕业于北京理工大学计算机学院，喜欢算法和数学等有趣的事情，热爱Python语言，有过Web和iOS开发经历。

TURING 图灵程序设计丛书

Python 语言及其应用

Introducing Python

[美] Bill Lubanovic 著
丁嘉瑞　梁杰　禹常隆 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（C I P）数据

Python语言及其应用 / （美）卢布诺维克
(Lubanovic,B.）著 ; 丁嘉瑞, 梁杰, 禹常隆译. -- 北
京 : 人民邮电出版社, 2016.1
（图灵程序设计丛书）
ISBN 978-7-115-40709-2

Ⅰ. ①P… Ⅱ. ①卢… ②丁… ③梁… ④禹… Ⅲ. ①
软件工具－程序设计 Ⅳ. ①TP311.56

中国版本图书馆CIP数据核字(2015)第245113号

内 容 提 要

本书介绍 Python 语言的基础知识及其在各个领域的具体应用，基于最新版本 3.x。书中首先介绍了 Python 语言的一些必备基本知识，然后介绍了在商业、科研以及艺术领域使用 Python 开发各种应用的实例。文字简洁明了，案例丰富实用，是一本难得的 Python 入门手册。

本书适合所有编程初学者阅读。

◆ 著　[美] Bill Lubanovic
译　丁嘉瑞　梁　杰　禹常隆
责任编辑　岳新欣
执行编辑　李艳玲　冯雪松
责任印制　杨林杰
◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷
◆ 开本：800×1000　1/16
印张：25.25
字数：630千字　2016年 1 月第 1 版
印数：1 – 5 000册　2016年 1 月北京第 1次印刷
著作权合同登记号　图字：01-2015-3677号

定价：79.00元
读者服务热线：(010)51095186转600　印装质量热线：(010)81055316
反盗版热线：(010)81055315
广告经营许可证：京崇工商广字第 0021 号

O’Reilly Media, Inc.介绍

O’Reilly Media 通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自 1978 年开始，O’Reilly 一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly 的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly 为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了 Make 杂志，从而成为 DIY 革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly 的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly 现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版、在线服务或者面授课程，每一项 O’Reilly 的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar 博客有口皆碑。”

——*Wired*

“O’Reilly 凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——*Business 2.0*

“O’Reilly Conference 是聚集关键思想领袖的绝对典范。”

——*CRN*

“一本 O’Reilly 的书就代表一个有用、有前途、需要学习的主题。”

——*Irish Times*

“Tim 是位特立独行的商人，他不光放眼于最长远、最广阔的视野，并且切实地按照 Yogi Berra 的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去，Tim 似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——*Linux Journal*

献给Mary、Karin、Tom和Roxie。

目录

前言

本书介绍 Python 编程语言，主要面向编程初学者。不过，如果你是一位有经验的程序员，想再学门 Python 编程语言，本书也很适合作为入门读物。

本书节奏适中，从基础开始逐步深入其他话题。我会结合食谱和教程的风格来解释新术语和新概念，但不会一次介绍很多。你会尽早并且常常接触到真实的 Python 代码。

虽然本书是入门读物，但我还是介绍了一些看起来比较高阶的话题，比如 NoSQL 数据库和消息传递库。之所以介绍它们，是因为在解决某类问题时它们比标准库更加合适。你需要下载并安装这些第三方 Python 包，从而更好地理解 Python“内置电池”适用于什么场景。此外，尝试新事物本身也充满乐趣。

我还会展示一些反面的例子，提醒你不要那么去做。如果你之前使用过其他语言并且想把风格照搬到 Python 的话，要格外注意。还有，我不认为 Python 是完美的，我会告诉你哪些东西应该避免。

> 书中有时会出现类似本条的提示内容，主要用于解释一些容易混淆的概念或者用更合适的 Python 风格的方法来解决同一个问题。

目标读者

本书的目标读者是那些对世界上最流行的计算语言感兴趣的人，无论你之前是否学过编程。

本书结构

本书前 7 章介绍 Python 基础知识，建议按顺序阅读。后面 5 章介绍如何在不同的应用场景中使用 Python，比如 Web、数据库、网络，等等，可以按任意顺序阅读。附录 A、B、C 介绍 Python 在艺术、商业和科学方面的应用，附录 D 是 Python 3 的安装教程，附录 E 和附录 F 是每章练习题的答案和速查表。

- 第 1 章

  程序和织袜子或者烤土豆很像。通过一些真实的 Python 程序可以了解这门语言的概貌、能力以及在真实世界中的用途。Python 和其他语言相比有很多优势，不过也有一些不完美的地方。旧版本的 Python（Python 2）正在被新版本（Python 3）替代。如果你在使用 Python 2，请安装 Python 3。你可以使用交互式解释器自行尝试本书中的代码示例。

- 第 2 章

  该章会介绍 Python 中最简单的数据类型：布尔值、整数、浮点数和文本字符串。你也会学习基础的数学和文本操作。

- 第 3 章

  该章会学习 Python 的高级内置数据结构：列表、元组、字典和集合。你可以像玩乐高积木一样用它们来构建更复杂的结构，并学到如何使用*迭代器*和*推导式*来遍历它们。

- 第 4 章

  该章会学习如何在之前学习的数据结构上用代码实现比较、选择和重复操作。你会学习如何用*函数*来组织代码，并用*异常*来处理错误。

- 第 5 章

  该章会介绍如何使用模块、包和程序组织大型代码结构。你会学习如何划分代码和数据、数据的输入输出、处理选项、使用 Python 标准库并了解标准库的内部实现。

- 第 6 章

  如果你已经在其他语言中学过面向对象编程，就可以轻松掌握 Python 的写法。该章会介绍对象和类的适用场景，有时候使用模块甚至列表和字典会更加合适。

- 第 7 章

  该章会介绍如何像专家一样处理数据。你会学到如何处理文本和二进制数据以及 Unicode 字符和 I/O。

- 第 8 章

  数据需要地方来存放。在该章中，你首先会学习使用普通文件、目录和文件系统，接着会学习如何处理常用文件格式，比如 CSV、JSON 和 XML。此外，你还会了解如何从关系型数据库甚至是最新的 NoSQL 数据库中存取数据。

- 第 9 章

  该章单独介绍 Web，包括客户端、服务器、数据抓取、API 和框架。你会编写一个带请求参数处理和模板的真实网站。

- 第 10 章

  该章会介绍系统相关内容，难度较高。你会学习如何管理程序、进程和线程，处理日期和时间，实现系统管理任务自动化。

- 第 11 章

  该章会介绍网络相关内容：服务、协议和 API。该章示例覆盖了底层 TCP 套接字、消

息库以及队列系统、云端部署。

- 第 12 章
  该章会介绍 Python 相关的小技巧，比如安装、使用 IDE、测试、调试、日志、版本控制和文档，还会介绍如何寻找并安装有用的第三方包、打包自己的代码以供重用，以及如何寻找更多有用的信息。祝你好运。
- 附录 A
  附录 A 会介绍 Python 在艺术领域的应用：图像、音乐、动画和游戏。
- 附录 B
  Python 在商业领域也有应用：数据可视化（图表、图形和地图）、安全和管理。
- 附录 C
  Python 在科学领域应用得尤其广泛：数学和统计学、物理科学、生物科学以及医学。附录 C 会介绍 NumPy、SciPy 和 Pandas。
- 附录 D
  如果你还没有安装 Python 3，附录 D 会介绍 Windows、Mac OS/X、Linux 和 Unix 下的安装方法。
- 附录 E
  附录 E 包含每章结尾的练习答案。请在亲自尝试解答之后再查看答案。
- 附录 F
  附录 F 包含一些有用的速查内容。

Python版本

开发者会不断向计算机语言中加入新特性、修复问题，因此计算机语言一直在变化。本书中的代码示例在 Python 3.3 中编写和测试。在本书编辑期间 Python 3.4 发布了，我会介绍一些新版本的内容。如果你想了解相关信息和特性的发布时间，可以阅读 What's New in Python 页面（https://docs.python.org/3/whatsnew/）。这个页面技术性比较强，对于 Python 初学者来说难度较大，不过如果你之后想研究 Python 的兼容性，可以阅读它。

排版约定

本书使用了下列排版约定。

- 楷体
  表示新术语。
- 等宽字体（`constant width`）
  表示程序片段，以及正文中出现的变量、函数名、数据库、数据类型、环境变量、语句和关键字等。

- 加粗等宽字体（**`constant width bold`**）
  表示应该由用户输入的命令或其他文本。

该图标表示一般注记。

该图标表示警告或警示。

使用代码示例

补充材料（代码示例、练习等）可以从 https://github.com/madscheme/introducing-python 下载。

本书是要帮你完成工作的。一般来说，如果本书提供了示例代码，你可以把它用在你的程序或文档中。除非你使用了很大一部分代码，否则无需联系我们获得许可。比如，用本书的几个代码片段写一个程序就无需获得许可，销售或分发 O'Reilly 图书的示例光盘则需要获得许可；引用本书中的示例代码回答问题无需获得许可，将书中大量的代码放到你的产品文档中则需要获得许可。

我们很希望但并不强制要求你在引用本书内容时加上引用说明。引用说明一般包括书名、作者、出版社和 ISBN。比如："*Introducing Python* by Bill Lubanovic(O'Reilly). Copyright 2015 Bill Lubanovic, 978-1-449-35936-2."

如果你觉得自己对示例代码的用法超出了上述许可的范围，欢迎你通过 permissions@oreilly.com 与我们联系。

Safari® Books Online

Safari Books Online（http://www.safaribooksonline.com）是应运而生的数字图书馆。它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。技术专家、软件开发人员、Web 设计师、商务人士和创意专家等，在开展调研、解决问题、学习和认证培训时，都将 Safari Books Online 视作获取资料的首选渠道。

对于组织团体、政府机构和个人，Safari Books Online 提供各种产品组合和灵活的定价策略。用户可通过一个功能完备的数据库检索系统访问 O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM

Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology 以及其他几十家出版社的上千种图书、培训视频和正式出版之前的书稿。要了解 Safari Books Online 的更多信息，我们网上见。

联系我们

请把对本书的评价和问题发给出版社。

美国：

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街 2 号成铭大厦 C 座 807 室（100035）
奥莱利技术咨询（北京）有限公司

O'Reilly 的每一本书都有专属网页，你可以在那儿找到本书的相关信息，包括勘误表、示例代码以及其他信息。本书的网站地址是：

http://shop.oreilly.com/product/0636920028659.do

对于本书的评论和技术性问题，请发送电子邮件到：

bookquestions@oreilly.com

要了解更多 O'Reilly 图书、培训课程、会议和新闻的信息，请访问以下网站：

http://www.oreilly.com

我们在 Facebook 的地址如下：

http://facebook.com/oreilly

请关注我们的 Twitter 动态：

http://twitter.com/oreillymedia

我们的 YouTube 视频地址如下：

http://www.youtube.com/oreillymedia

致谢

非常感谢那些阅读本书初稿并给予反馈的人。我尤其要感谢仔细审阅本书的 Eli Bessert、Henry Canival、Jeremy Elliott、Monte Milanuk、Loïc Pefferkorn 和 Steven Wayne。

第 1 章

Python初探

我们从一个小谜题以及它的答案开始。你认为下面这两行的含义是什么？

```
(Row 1): (RS) K18,ssk,k1,turn work.
(Row 2): (WS) Sl 1 pwise,p5,p2tog,p1,turn.
```

它们看起来像是某种计算机程序。实际上，这是一个针织图案。更准确地说，这两行描述的是如何编织袜子的足跟部分。对我来说，看懂它们就像让猫看懂《纽约时报》上的填字游戏一样难，但是对我妻子来说却轻而易举。如果你也懂编织，一样可以轻松看懂。

来看另一个例子。虽然你不知道最终会做出什么，但是马上就能明白下面的内容是什么。

```
    1/2杯黄油或者人造黄油
    1/2杯奶油
    2.5杯面粉
    1茶匙盐
    1汤匙糖
    4杯糊状土豆(冷藏)

确保在加入面粉之前冷藏所有材料。
混合所有材料。
用力揉。
揉成20个球并冷藏。
对于每一个球：
    在布上洒上面粉。
    用擀面杖把球擀成圆饼。
    入锅,炸至棕色。
    翻面继续炸。
```

即使你不会做饭，应该也能看懂这是一个菜谱：一系列食物原料以及准备工作。这道菜是什么呢？是 lefse，一道和玉米饼很像的挪威美食。做好之后抹上黄油、果酱或者其他你喜欢吃的东西，最后卷起来吃。

编织图案和菜谱有一些共同的特征。

- 专有名词、缩写以及符号。有些很常见，有些很难懂。
- 规定专有名词、缩写以及符号的使用方法，也就是它们的*语法*。
- 一个操作序列，按照顺序进行。
- 有时需要重复一些操作（*循环*），比如炸 lefse 的每一面。
- 有时需要引用其他操作序列（用计算机术语来说就是一个*函数*）。在菜谱中，你可能需要引用另一个将土豆捣成糊状的菜谱。
- 假定已经有相关知识。菜谱假定你知道水是什么以及如何烧水。编织图案假定你学过编织并且不会经常扎到手。
- 一个期望的结果。在我们的例子中分别是袜子和食物，千万不要把它们混在一起哦。

以上这些概念都会出现在计算机程序中。这两个例子的目的是让你知道编程并不像看起来那么高深莫测，其实只是学习一些正确的单词和规则而已。

下面来看看真正的程序。你知道它在做什么吗？

```
for countdown in 5, 4, 3, 2, 1, "hey!":
    print(countdown)
```

这其实是一段 Python 程序，会打印出下面的内容：

```
5
4
3
2
1
hey!
```

看到了吗？学习 Python 就像看懂菜谱或者编织图案一样简单。此外，你可以在桌子上舒服并且安全地练习编写 Python 程序，完全不用担心被热水烫到或者被针扎到。

Python 程序有一些特殊的单词和符号——`for`、`in`、`print`、逗号、冒号、括号以及其他符号。这些单词和符号是语法的重要组成部分。好消息是，Python 的语法非常优秀，相比其他大多数编程语言，学习 Python 需要记住的语法内容很少。Python 语法非常自然，就像一份菜谱一样。

下面的 Python 程序会从一个 Python *列表*（list）中选出一条电视新闻的常用语并打印出来：

```
cliches = [
    "At the end of the day",
    "Having said that",
    "The fact of the matter is",
    "Be that as it may",
    "The bottom line is",
    "If you will",
    ]
print(cliches[3])
```

程序会打印出第四条常用语：

```
Be that as it may
```

一个 Python 列表，比如 `cliches`，就是一个值序列，可以通过它们相对于列表起始位置的*偏移量*来访问。第一个值的偏移量是 `0`，第四个值的偏移量是 `3`。

> 人们通常从 1 开始数数，所以从 0 开始数似乎很奇怪。用偏移量来代替位置会更好理解一些。

列表在 Python 中很常用，第 3 章会讲解列表的用法。

下面这段程序同样会打印出一条引用内容，但是这次是用说话者的人名而不是列表中的位置来进行访问：

```
quotes = {
    "Moe": "A wise guy, huh?",
    "Larry": "Ow!",
    "Curly": "Nyuk nyuk!",
    }
stooge = "Curly"
print(stooge, "says:", quotes[stooge])
```

运行这个小程序会打印出：

```
Curly says: Nyuk nyuk!
```

`quotes` 是一个 Python *字典*。字典是一个集合，包含唯一*键*（本例中是跟屁虫“Stooge”的名字）及其关联的*值*（本例中是跟屁虫说的话）。使用字典可以通过名字来存储和查找东西，和列表一样非常有用。第 3 章会详细讲解字典。

常用语的例子中使用方括号（`[` 和 `]`）来创建 Python 列表，跟屁虫的例子中使用大括号（`{` 和 `}`，大括号的英文是 curly bracket，但是大括号和 Curly 没有任何关系[1]）来创建 Python 字典。这些都是 Python 的*语法*，在之后的内容中你会看到更多语法。

现在我们来看另一个完全不同的例子：示例 1-1 中的 Python 程序会执行一系列复杂的任务。你可能还看不懂这段程序，没关系，学完本书之后就可以看懂了。这个例子的目的是让你了解典型的 Python 程序长什么样。如果你了解其他计算机语言，可以对比一下。

示例 1-1 会连接 YouTube 网站并获取当前评价最高的视频的信息。如果 YouTube 返回的是常见的 HTML 文本，那就很难从中挖掘出我们想要的信息（9.3.4 节会介绍*网页抓取*）。幸运的是，它返回的是 JSON 格式的数据，这种格式可以直接用计算机处理。JSON（JavaScript Object Notation，JavaScript 对象符号）是一种人类可以阅读的文本格式，它描述了类型、值以及值的顺序。JSON 就像一个小型编程语言，使用 JSON 在不同计算机语言和系统之间交换数据已经成为了一种非常流行的方式。更多关于 JSON 的内容请参考 8.2.4 节。

注 1：Curly 是美国乌鸦童子军（Crow Scouts）的一员，乌鸦童子军是美国和印第安人打仗时由印第安人战俘组成的军队。Curly 是小巨角河战役中为数不多的幸存者之一。小巨角河战役是美军和北美势力最庞大的苏族印地安人之间的战争，在这场战争中印第安人歼灭了美国历史上最有名的第七骑兵团，Curly 当时没有参战，他是第一个报告第七骑兵团战败的人，也因此出名。——译者注

Python 程序可以把 JSON 文本翻译成 Python 的数据结构——你会在之后的两章中学习它们——和你自己创建出来的一样。这个 YouTube 响应包含很多数据，作为演示我只打印出了前 6 个视频的标题。再说一次，这是一个完整的 Python 程序，你自己也可以运行一下。

示例 1-1：intro/youtube.py

```
import json
from urllib.request import urlopen
url = "https://gdata.youtube.com/feeds/api.standardfeeds/top_rated?alt=json"
response = urlopen(url)
contents = response.read()
text = contents.decode('utf8')
data = json.loads(text)
for video in data['feed']['entry'][0:6]:
    print(video['title']['$t'])
```

最后一次运行这个程序得到的输出是：

```
Evolution of Dance - By Judson Laipply
Linkin Park - Numb
Potter Puppet Pals: The Mysterious Ticking Noise
"Chocolate Rain" Original Song by Tay Zonday
Charlie bit my finger - again !
The Mean Kitty Song
```

这个 Python 小程序仅仅用了 9 行代码就很好地完成了任务，并且具备很高的可读性。如果你看不懂下面的术语，没关系，接下来的几章会让你明白它们的意思。

- 第 1 行：从 Python 标准库中导入名为 `json` 的所有代码。
- 第 2 行：从 Python 标准 `urllib` 库中导入 `urlopen` 函数。
- 第 3 行：给变量 `url` 赋值一个 YouTube 地址。
- 第 4 行：连接指定地址处的 Web 服务器并请求指定的 Web 服务。
- 第 5 行：获取响应数据并赋值给变量 `contents`。
- 第 6 行：把 `contents` 解码成一个 JSON 格式的文本字符串并赋值给变量 `text`。
- 第 7 行：把 `text` 转换为 `data`——一个存储视频信息的 Python 数据结构。
- 第 8 行：每次获取一个视频的信息并赋值给变量 `video`。
- 第 8 行：使用两层 Python 字典（`data['feed']['entry']`）和切片操作（`[0:6]`）。
- 第 9 行：使用 `print` 函数打印出视频标题。

视频信息中包含多种你之前见过的 Python 数据结构，第 3 章会详细介绍。

在这个例子中，我们使用了一些 Python 标准库模块（它们是安装 Python 时就已经包含的程序），但是它们并不是最好的。下面的代码使用第三方 Python 软件包 `requests` 重写了这个例子：

```
import requests
url = "https://gdata.youtube.com/feeds/api.standardfeeds/top_rated?alt=json"
response = requests.get(url)
data = response.json()
for video in data['feed']['entry'][0:6]:
    print(video['title']['$t'])
```

新版代码只有 6 行，并且我认为可读性更高。第 5 章会详细介绍 `requests` 以及其他第三方 Python 软件。

1.1 真实世界中的Python

那么，是否真的值得付出时间和努力来学习 Python 呢？它真的有用吗？实际上，Python 诞生于 1991 年（比 Java 还早），并且一直是最流行的十门计算机语言之一。公司需要雇用程序员来写 Python 程序，包括你每天都会用到的 Google、YouTube、Dropbox、Netflix 和 Hulu 等。我用 Python 开发过许多产品级应用，从邮件搜索应用到商业网站都有。对于发展迅速的组织来说，Python 能极大地提高生产力。

Python 可以应用在许多计算环境下，如下所示：

- 命令行窗口
- 图形用户界面，包括 Web
- 客户端和服务端 Web
- 大型网站后端
- 云（第三方负责管理的服务器）
- 移动设备
- 嵌入式设备

Python 程序从一次性*脚本*——就像你在本章中看到的一样——到几十万行的系统都有。我们会介绍 Python 在网站、系统管理和数据处理方面的应用，还会介绍 Python 在艺术、科学和商业方面的应用。

1.2 Python与其他语言

Python 和其他语言相比如何呢？什么时候该选择什么语言呢？本节会展示一些其他语言的代码片段，这样更直观一些。如果有些语言你从未使用过，也不必担心，你并不需要看懂所有代码（当你看到最后的 Python 示例时，会发现没学过其他语言也不是什么坏事）。如果你只对 Python 感兴趣，完全可以跳过这一节。

下面的每段程序都会打印出一个数字和一条描述语言的信息。

如果你使用的是命令行或者终端窗口，那你使用的就是 shell 程序，它会读入你的命令、运行并显示结果。Windows 的 shell 叫作 `cmd`，它会运行后缀为 .bat 的 batch 文件。Linux 和其他类 Unix 系统（包括 Mac OS X）有许多 shell 程序，最流行的称为 bash 或者 `sh`。shell 有许多简单的功能，比如执行简单的逻辑操作以及把类似 `*` 的通配符扩展成文件名。你可以把命令保存到名为“shell 脚本”的文件中稍后运行。shell 可能是程序员接触到的第一个程序。它的问题在于程序超过百行之后扩展性很差，并且比其他语言的运行速度慢很多。下面就是一段 shell 程序：

```
#!/bin/sh
language=0
echo "Language $language: I am the shell. So there."
```

如果你把这段代码保存为 meh.sh 并通过 sh meh.sh 命令来运行它，就会看到下面的输出：

```
Language 0: I am the shell. So there.
```

老牌语言 C 和 C++ 是底层语言，只有极其重视性能时才会使用。它们很难学习，并且有许多细节需要你自己处理，处理不当就可能导致程序崩溃和其他很难解决的问题。下面是一段 C 程序：

```
#include <stdio.h>
int main(int argc, char *argv[]) {
    int language = 1;
    printf("Language %d: I am C! Behold me and tremble!\n", language);
    return 0;
}
```

C++ 和 C 看起来很相似，但是特性完全不同：

```
#include <iostream>
using namespace std;
int main(){
    int language = 2;
    cout << "Language " << language << \
        ": I am C++! Pay no attention to that C behind the curtain!" << \
        endl;
    return(0);
}
```

Java 和 C# 是 C 和 C++ 的接班人，解决了后者的许多缺点，但是相比之下代码更加冗长，写起来也有许多限制。下面是 Java 代码：

```
public class Overlord {
    public static void main (String[] args) {
        int language = 3;
        System.out.format("Language %d: I am Java! Scarier than C!\n", language);
    }
}
```

如果你没写过这些语言的程序，可能会觉得很奇怪：这都是什么东西？有些语言有很大的语法包袱。它们有时被称为*静态语言*，因为你必须告诉计算机许多底层细节，下面我来解释一下。

语言有*变量*——你想在程序中使用的值的名字。静态语言要求你必须声明每个变量的*类型*：它会使用多少内存以及允许的使用方法。计算机利用这些信息把程序*编译*成非常底层的*机器语言*（专门给计算机硬件使用的语言，硬件很容易理解，但是人类很难理解）。计算机语言的设计者通常必须进行权衡，到底是让语言更容易被人使用还是更容易被计算机使用。声明变量类型可以帮助计算机发现更多潜在的错误并提高运行速度，但是却需要使用者进行更多的思考和编程。C、C++ 和 Java 代码中经常需要声明类型。举例来说，在上面的例子中必须使用 int 将 language 变量声明为一个整数。（其他类型的存储方式和整数不同，比如浮点数 3.14159、字符以及文本数据。）

那么为什么它们被称为静态语言呢？因为这些语言中的变量不能改变类型。它们是静态的。整数就是整数，永远无法改变。

相比之下，*动态语言*（也被称为*脚本语言*）并不需要在使用变量前进行声明。假设你输入 `x = 5`，动态语言知道 5 是一个整数，因此变量 x 也是整数。这些语言允许你用更少的代码做更多的事情。动态语言的代码不会被编译，而是由*解释器*程序来*解释*执行。动态语言通常比编译后的静态语言更慢，但是随着解释器的不断优化，动态语言的速度也在不断提升。长期以来，动态语言的主要应用场景都是很短的程序（*脚本*），比如给静态语言编写的程序进行数据预处理。这样的程序通常称为*胶水代码*。虽然动态语言很擅长做这些事，但是如今它们也已经具备了处理大型任务的能力。

许多年来，Perl（http://www.perl.org/）一直是一门万能的动态语言。Perl 非常强大并且有许多扩展库。然而，它的语法非常难用，并且似乎无法阻挡 Python 和 Ruby 的崛起。下面是一段 Perl 代码：

```
my $language = 4;
print "Language $language: I am Perl, the camel of languages.\n";
```

Ruby（http://www.ruby-lang.org/）是一门新语言。它借鉴了一些 Perl 的特点，并且因为 Web 开发框架 Ruby on Rails 红遍大江南北。Ruby 和 Python 的许多应用场景相同，选择哪一个通常看个人喜好或者是否有你需要的库。下面是一段 Ruby 代码：

```
language = 5
puts "Language #{language}: I am Ruby, ready and aglow."
```

PHP（http://www.php.net/）在 Web 开发领域非常流行，因为它可以轻松结合 HTML 和代码，就像例子中展示的那样。然而，PHP 语言本身有许多缺陷，并且很少被应用在 Web 以外的领域。

```
<?PHP
$language = 6;
echo "Language $language: I am PHP. The web is <i>mine<i>, I say.\n";
?>
```

最后是我们的主角，Python：

```
language = 7
print("Language %s: I am Python. What's for supper?" % language)
```

1.3 为什么选择Python

Python 是一门非常通用的高级语言。它的设计极大地增强了代码可读性，可读性远比听上去重要得多。每个计算机程序只被编写一次，但是会被许多人阅读和修改许多次。提高可读性也可以让学习和记忆更加容易，因此也更容易修改。和其他流行的语言相比，Python 的学习曲线更加平缓，可以让你很快具备生产力，当然，想成为专家还需要深入学习才行。

Python 简洁的语法可以让你写出比静态语言更短的程序。研究证明，程序员每天可以编写的代码行数是有限的——无论什么语言，因此，如果完成同样的功能只需要编写一半长度的代码，生产力就可以提高一倍。对于重视这一点的公司来说，Python 是一个不算秘密的秘密武器。

在顶尖的美国大学中（http://cacm.acm.org/blogs/blog-cacm/176450-python-is-now-the-most-popular-introductory-teaching-language-at-top-us-universities/fulltext），Python 是计算机入门课程中最流行的语言。此外，它也被两千多名雇主（http://blog.codeeval.com/codeevalblog/2014#.U73vaPldUpw=）用来评估编程技能。

当然，它是免费的，就像啤酒和演讲一样。你可以免费用 Python 来编写任何东西并用在任何地方。没人可以一边阅读你的 Python 程序一边说："这是一个非常棒的小程序，希望不会发生什么意外。"

Python 几乎可以运行在任何地方并且其标准库中有很多有用的软件。

不过，选择 Python 最关键的理由可能出乎你的意料：大家都喜欢它。实际上，大家不只是把 Python 当作一个完成工作的工具，而是非常享受用它编程。在工作中不得不用其他语言时，人们通常会非常想念 Python 的某些特性。这就是 Python 能够胜出的原因。

1.4　何时不应该使用Python

Python 并非在所有场合都是最好用的语言。

它并不是默认安装在所有环境中。如果你的电脑上没有 Python，附录 D 会告诉你如何安装。

对于大多数应用来说，Python 已经足够快了，但是有些场合下，它的性能仍然是个问题。如果你的程序会花费大量时间用于计算（专业术语是*中央处理器受限*），那么可以使用 C、C++ 或者 Java 来编写程序从而提高性能。但是这并不是唯一的选择！

- 有时候用 Python 实现一个更好的*算法*（一系列解决问题的步骤）可以打败 C 中的低效算法。Python 对于开发效率的提升可以让你有更多的时间来尝试各种选择。
- 在许多应用中，程序会因为等待其他服务器的响应而浪费时间。这段时间里 CPU（中央处理单元，计算机中负责所有计算的*芯片*）几乎什么都不做，因此，静态和动态程序的端到端时间几乎是一样的。
- Python 的标准解释器用 C 实现，所以可以通过 C 代码进行扩展。12.10 节会介绍一些。
- Python 解释器变得越来越快。Java 最初也很慢，经过大量的研究和资金投入之后，它变得非常快。Python 并不属于某个公司，因此它的发展会更缓慢一些。12.10.4 节会介绍 PyPy 项目及其意义。
- 可能你的项目要求非常严格，无论如何努力 Python 都无法达到要求。那么，借用伊恩·荷姆在电影《异形》中说过的一句话，我很同情你。通常来说可以选择 C、C++ 和 Java，不过新语言 Go（http://golang.org，写起来像 Python，性能像 C）也是一个不错的选择。

1.5　Python 2与Python 3

你即将面临的最大问题是，Python 有两个版本。Python 2 已经存在了很长时间并且预装在 Linux 和 Apple 电脑中。Python 是一门很出色的语言，但是世界上不存在完美的东西。和

其他领域一样，在计算机语言中许多问题很容易解决，但是也有一些问题很难解决。后者的难点在于不兼容：使用修复后的新版本编写的程序无法运行在旧的 Python 系统中，旧的程序也无法运行在新的系统中。

Python 的发明者（吉多 · 范 · 罗苏姆，https://www.python.org/~guido）和其他开发者决定把这些困难问题放在一起解决，并把解决后的版本称作 Python 3。Python 2 已经成为过去，Python 3 才是未来。Python 2 的最后一个版本是 2.7，它会被支持很长一段时间，但也就仅此而已，再也没有 Python 2.8 了。新的开发全部会在 Python 3 上进行。

本书使用的是 Python 3。如果你使用的是 Python 2 也不用担心，两者差别不大。最明显的区别在于调用 `print` 的方式，最重要的区别则是处理 Unicode 字符的方式，详情参见第 2 章和第 7 章。流行的 Python 软件需要逐步升级，和常见的“先有鸡还是先有蛋”问题一样。不过，看起来我们现在终于到达了发生转变的临界点。

1.6　安装Python

为了让这章更加简洁，安装 Python 3 的细节参见附录 D。如果你还没安装 Python 3 或者不确定是否安装过 Python，请阅读附录 D。

1.7　运行Python

安装好 Python 3 之后，可以用它来运行本书中的 Python 程序和你自己的 Python 代码。那么如何运行 Python 程序呢？通常来说有两种方法。

- Python 自带的*交互式解释器*可以很方便地执行小程序。你可以一行一行输入命令然后立刻查看运行结果。这种方式可以很好地结合输入和查看结果，从而快速进行一些实验。我会用交互式解释器来说明一些语言特性，你可以在自己的 Python 环境中输入同样的命令。
- 除此之外，可以把 Python 程序存储到文本文件中，通常要加上 .py 扩展名，然后输入 `python` 加文件名来执行。

我们来分别尝试一下这两种方式。

1.7.1　使用交互式解释器

本书中大多数代码示例都用到了交互式解释器。当你输入示例中的命令并且看到同样的输出时，就可以确定你没有跑偏。

可以在电脑上输入 Python 主程序的名称来启动解释器：应该是 `python`、`python3` 或者类似的东西。在本书接下来的内容中，我们会假设它叫 `python`；如果你的 Python 主程序名字不同，请把代码示例中的 `python` 全部替换成你电脑上的名称。

交互式解释器的工作原理基本上和 Python 对文件的处理方式一样，除了一点：当你输入一些包含值的东西时，交互式解释器会自动打印出这个值。举例来说，如果你启动 Python 并输入数字 `61`，它会立刻出现在终端中。

在下面的例子中，$ 表示系统提示符，用来输入终端中的命令，比如 python。本书的代码示例都会使用它，尽管在你的电脑中提示符可能不是 $。

```
$ python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 61
61
>>>
```

这种自动打印值的省时特性只有交互式解释器中有，在 Python 语言中没有。

顺便说一句，也可以使用 print() 在解释器中打印内容：

```
>>> print(61)
61
>>>
```

如果你亲自动手在交互式解释器中执行了上面这些例子并看到了相同的结果，恭喜你，你成功运行了真正的 Python 代码（虽然有点短）。接下来的几章中会接触到更长的 Python 程序。

1.7.2　使用Python文件

如果你把 61 放在文件中并使用 Python 来执行它，确实可以，但是程序什么都不会输出。在非交互式的 Python 程序中，你必须调用 print 函数来打印内容，如下所示：

```
print(61)
```

我们来生成一个 Python 程序文件并运行它。

(1) 打开你的文本编辑器。
(2) 输入代码 print(61)，如上所示。
(3) 保存文件，命名为 61.py。一定要确保存储为纯文本而不是“富文本”格式，比如 RTF 或者 Word。Python 程序文件并不是一定要以 .py 结尾，但是加上它可以让你清楚这个文件的作用。
(4) 如果你使用的是图形用户界面——基本上每个人都会用——请打开一个终端窗口[2]。
(5) 输入下面的命令来运行程序：

```
$ python 61.py
```

应该可以看到一行输出：

```
61
```

注 2：如果你不明白什么是终端窗口，请阅读附录 D。

成功了吗？如果成功了，恭喜你运行了你的第一个 Python 程序！

1.7.3　下一步

你可以向真正的 Python 系统中输入命令，不过它们必须符合 Python 的语法。我们不会一次性向你展示所有的语法规则，而是在接下来的几章中逐一进行讲解。

开发 Python 程序最基础的方法是使用一个纯文本编辑器和一个终端窗口。在本书中会直接展示纯文本，有时候在交互式终端中，有时候在 Python 文件中。不过除此之外，还有很多优秀的 Python *集成开发环境*（IDE）。它们的图形用户界面可能会有许多高端文本编辑功能和辅助开发功能。在第 12 章中你会看到一些相关介绍。

1.8　禅定一刻

每种计算机语言都有自己的风格。在前言中我提到过，你可以用 Python 的方式来表达自己。Python 中内置了一些自由体诗歌，它们简单明了地说明了 Python 的哲学（就我所知，Python 是唯一一个包含这种复活节彩蛋的语言）。只要在交互式解释器中输入 `import this`，然后按下回车就能看到它们：

```
>>> import this
《Python之禅》   Tim Peters

优美胜于丑陋
明了胜于隐晦
简洁胜于复杂
复杂胜于混乱
扁平胜于嵌套
宽松胜于紧凑
可读性很重要
即便是特例,也不可违背这些规则
虽然现实往往不那么完美
但是不应该放过任何异常
除非你确定需要如此
如果存在多种可能,不要猜测
肯定有一种——通常也是唯一一种——最佳的解决方案
虽然这并不容易,因为你不是Python之父
动手比不动手要好
但不假思索就动手还不如不做
如果你的方案很难懂,那肯定不是一个好方案
如果你的方案很好懂,那肯定是一个好方案
命名空间非常有用,我们应当多加利用
```

这些哲学观点会贯穿全书。

1.9　练习

本章介绍了 Python 语言——它是干什么的、它是什么样的以及它在计算机世界中的作用。在每章的结尾我都会列出一些小练习来帮助你巩固刚学到的知识并为学习新知识做好准备。

(1) 如果你还没有安装 Python 3，现在就立刻动手。具体方法请阅读附录 D。

(2) 启动 Python 3 交互式解释器。再说一次，具体方法请阅读附录 D。它会打印出几行信息和一行 `>>>`，这是你输入 Python 命令的提示符。

(3) 随便玩玩解释器。可以用它来计算 `8 * 9`，按下回车来查看结果，Python 应该会打印出 `72`。

(4) 输入数字 `47` 并按下回车，解释器有没有在下一行打印出 `47`？

(5) 现在，输入 `print(47)` 并按下回车，解释器有没有在下一行打印出 `47`？

第 2 章

Python基本元素：数字、字符串和变量

本章会从 Python 最基本的内置数据类型开始学习，这些类型包括：

- 布尔型（表示真假的类型，仅包含 `True` 和 `False` 两种取值）
- 整型（整数，例如 `42`、`100000000`）
- 浮点型（小数，例如 `3.14159`，或用科学计数法表示的数字，例如 `1.0e8`，它表示 1 乘以 10 的 8 次方，也可写作 `100000000.0`）
- 字符串型（字符组成的序列）

这些基本类型就像组成 Python 的原子一样。本章将学习如何单独使用这些基本“原子”，而第 3 章则会介绍如何将这些“原子”组合成更大的“分子”。

每一种类型都有自己的使用规则，计算机对它们的处理方式也不尽相同。此外我们还会学到变量的概念（与实际数据相关联的名字，后面有更详细的介绍）。

本章中的代码虽然都只是小的片段，但它们都是有效的 Python 程序。为了方便快速测试，我们将使用 Python 的交互式解释器，这样在输入代码的同时就可以获得执行结果。尝试在你自己搭建的 Python 环境中运行书中的代码片段，它们会由提示符 `>>>` 标出。第 4 章将开始编写能独立执行的 Python 程序。

2.1　变量、名字和对象

Python 里所有数据——布尔值、整数、浮点数、字符串，甚至大型数据结构、函数以及程序——都是以对象（object）的形式存在的。这使得 Python 语言具有很强的统一性（还有许多其他有用的特性），而这恰恰是许多其他语言所缺少的。

对象就像一个塑料盒子，里面装的是数据（图 2-1）。对象有不同类型，例如布尔型和整

型，类型决定了可以对它进行的操作。现实生活中的“陶器”会暗含一些信息（例如它可能很重，注意不要掉到地上，等等）。类似地，Python 中一个类型为 `int` 的对象会告诉你：可以把它与另一个 `int` 对象相加。

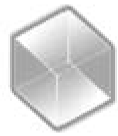


图 2-1：对象就像一个盒子

对象的类型还决定了它装着的数据是允许被修改的变量（*可变的*）还是不可被修改的常量（*不可变的*）。你可以把不可变对象想象成一个透明但封闭的盒子：你可以看到里面装的数据，但是无法改变它。类似地，可变对象就像一个开着口的盒子，你不仅可以看到里面的数据，还可以拿出来修改它，但你无法改变这个盒子本身，即你无法改变对象的类型。

Python 是*强类型的*（strongly typed），你永远无法修改一个已有对象的类型，即使它包含的值是可变的（图 2-2）。

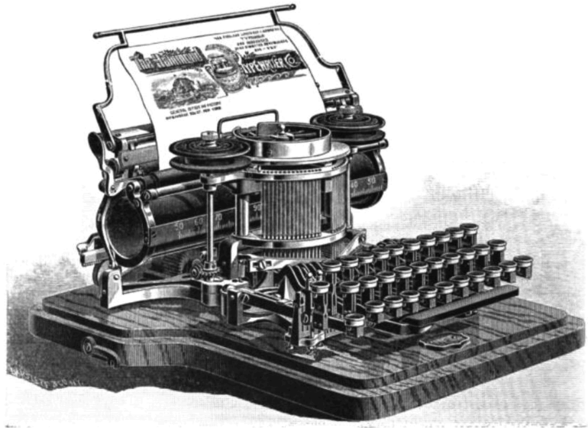


图 2-2：Strong Typing 不是指用力敲打键盘

编程语言允许你定义*变量*（variable）。所谓变量就是在程序中为了方便地引用内存中的值而为它取的名称。在 Python 中，我们用 = 来给一个变量赋值。

> 我们都在数学课上学过 = 代表“等于”，那么为什么计算机语言（包括 Python）要用 = 代表赋值操作呢？一种解释是标准键盘没有像左箭头一样的逻辑上能代表赋值操作的键，与其他键相比，= 显得相对不那么令人困惑；而在程序中，赋值出现的频率又要远远超过等于，因此把 = 分给了赋值操作来使用。

下面这段仅两行的 Python 程序首先将整数 7 赋值给了变量 a，之后又将 a 的值打印了出来：

```
>>> a = 7
>>> print(a)
7
```

注意，Python 中的变量有一个非常重要的性质：它仅仅是一个名字。赋值操作并不会实际复制值，它只是为数据对象取个相关的名字。名字是对对象的引用而不是对象本身。你可以把名字想象成贴在盒子上的标签（见图 2-3）。

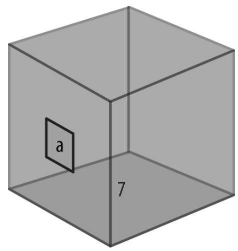


图 2-3：贴在对象上的名字

试着在交互式解释器中执行下面的操作：

(1) 和之前一样，将 7 赋值给名称 a，这样就成功创建了一个包含整数 7 的对象；
(2) 打印 a 的值；
(3) 将 a 赋值给 b，这相当于给刚刚创建的对象又贴上了标签 b；
(4) 打印 b 的值。

```
>>> a = 7
>>> print(a)
7
>>> b = a
>>> print(b)
7
```

在 Python 中，如果想知道一个对象（例如一个变量或者一个字面值）的类型，可以使用语句：`type( thing )`。试试对不同的字面值（`58`、`99.9`、`abc`）以及不同的变量（`a`、`b`）执行 `type` 操作：

```
>>> type(a)
<class 'int'>
>>> type(b)
<class 'int'>
>>> type(58)
<class 'int'>
>>> type(99.9)
<class 'float'>
>>> type('abc')
<class 'str'>
```

类（class）是对象的定义，第 6 章会详细介绍。在 Python 中，“类”和“类型”一般不加

区分。

变量名只能包含以下字符：

- 小写字母（a~z）
- 大写字母（A~Z）
- 数字（0~9）
- 下划线（_）

名字不允许以数字开头。此外，Python 中以下划线开头的名字有特殊的含义（第 4 章会解释）。下面是一些合法的名字：

- a
- a1
- a_b_c___95
- _abc
- _1a

下面这些名字则是非法的：

- 1
- 1a
- 1_

最后要注意的是，不要使用下面这些词作为变量名，它们是 Python 保留的关键字：

```
False      class      finally    is         return
None       continue   for        lambda     try
True       def        from       nonlocal   while
and        del        global     not        with
as         elif       if         or         yield
assert     else       import     pass
break      except     in         raise
```

这些关键字以及其他的一些标点符号是用于描述 Python 语法的。在本书中，你会慢慢学到它们各自的作用。

2.2 数字

Python 本身支持整数（比如 5 和 1000000000）以及浮点数（比如 3.1416、14.99 和 1.87e4）。你可以对这些数字进行下表中的计算。

运算符	描述	示例	运算结果
+	加法	5 + 8	13
-	减法	90 - 10	80
*	乘法	4 * 7	28
/	浮点数除法	7 / 2	3.5
//	整数除法	7 // 2	3

（续）

运算符	描述	示例	运算结果
%	模（求余）	7 % 3	1
**	幂	3 ** 4	81

接下来会给你展示一些示例，这些示例体现了 Python 作为一个计算机器的非凡特性。

2.2.1　整数

任何仅含数字的序列在 Python 中都被认为是整数：

```
>>> 5
5
```

你可以单独使用数字零（0）：

```
>>> 0
0
```

但不能把它作为前缀放在其他数字前面：

```
>>> 05
  File "<stdin>", line 1
    05
     ^
SyntaxError: invalid token
```

这是你第一次看见 Python 异常——程序错误。在上面的例子中，解释器抛出了一个警告，提示 05 是一个“非法标识”（invalid token）。2.2.3 节会解释这个警告的意义。你会在本书中见到许多种异常，这是 Python 主要的错误处理机制。

一个数字序列定义了一个正整数。你也可以显式地在前面加上正号 +，这不会使数字发生任何改变：

```
>>> 123
123
>>> +123
123
```

在数字前添加负号 - 可以定义一个负数：

```
>>> -123
-123
```

你可以像使用计算器一样使用 Python 来进行常规运算。Python 支持的运算参见之前的表格。试试进行加法和减法运算，运算结果和你预期的一样：

```
>>> 5 + 9
14
>>> 100 - 7
93
```

```
>>> 4 - 10
-6
```

可以连续运算任意个数：

```
>>> 5 + 9 + 3
17
>>> 4 + 3 - 2 - 1 + 6
10
```

格式提示：数字和运算符之间的空格不是强制的，你也可以写成下面这种格式：

```
>>> 5+9   +      3
17
```

只不过添加空格会使代码看起来更规整更便于阅读。

乘法运算的实现也很直接：

```
>>> 6 * 7
42
>>> 7 * 6
42
>>> 6 * 7 * 2 * 3
252
```

除法运算比较有意思，可能与你预期的有些出入，因为 Python 里有两种除法：

- / 用来执行浮点除法（十进制小数）
- // 用来执行整数除法（整除）

与其他语言不同，在 Python 中即使运算对象是两个整数，使用 / 仍会得到浮点型的结果：

```
>>> 9 / 5
1.8
```

使用整除运算得到的是一个整数，余数会被截去：

```
>>> 9 // 5
1
```

如果除数为 0，任何一种除法运算都会产生 Python 异常：

```
>>> 5 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 7 // 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by z
```

之前的例子中我们都在使用立即数进行运算，你也可以在运算中将立即数和已赋值过的变量混合使用：

```
>>> a = 95
>>> a
```

```
95
>>> a - 3
92
```

上面代码中出现了 `a - 3`，但我们并没有将结果赋值给 a，因此 a 的值并未发生改变：

```
>>> a
95
```

如果你想要改变 a 的值，可以这样写：

```
>>> a = a - 3
>>> a
92
```

对于初学者来说，上面这行式子可能很费解，因为小学的数学知识让我们根深蒂固地认为 = 代表等于，因此上面的式子显然是不成立的。但在 Python 里并非如此，Python 解释器会首先计算 = 右侧的表达式，然后将其结果赋值给左侧的变量。

试着这样理解看看是否有帮助。

- 计算 a-3
- 将运算结果保存在一个临时变量中
- 将这个临时变量的值赋值给 a：

```
>>> a = 95
>>> temp = a - 3
>>> a = temp
```

因此，当输入：

```
>>> a = a - 3
```

Python 实际上先计算了右侧的减法，暂时记住运算结果，然后将这个结果赋值给了 = 左侧的 a。这种写法比使用临时变量要更加迅速、简洁。

你还可以进一步将运算过程与赋值过程进行合并，只需将运算符放到 = 前面。例如，`a -= 3` 等价于 `a = a - 3`：

```
>>> a = 95
>>> a -= 3
>>> a
92
```

下面的代码等价于执行 `a = a + 8`：

```
>>> a += 8
>>> a
100
```

以此类推，下面的代码等价于 `a = a * 2`：

```
>>> a *= 2
>>> a
200
```

再试试浮点型除法，例如 a = a / 3：

```
>>> a /= 3
>>> a
66.66666666666667
```

接着将 13 赋值给 a，然后试试执行 a = a // 4（整数除法）的简化版：

```
>>> a = 13
>>> a //= 4
>>> a
3
```

百分号 % 在 Python 里有多种用途，当它位于两个数字之间时代表求模运算，得到的结果是第一个数除以第二个数的余数：

```
>>> 9 % 5
4
```

使用下面的方法可以同时得到余数和商：

```
>>> divmod(9,5)
(1, 4)
```

或者你也可以分别计算：

```
>>> 9 // 5
1
>>> 9 % 5
4
```

上面的代码出现了一些你没见过的新东西：一个叫作 divmod 的*函数*。这个函数接受了两个整数：9 和 5，并返回了一个包含两个元素的结果，我们称这种结构为*元组*（tuple）。第 3 章会学习元组的使用，而关于函数的内容将在第 4 章进行讲解。

2.2.2　优先级

想一想下面的表达式会产生什么结果？

```
>>> 2 + 3 * 4
```

如果你先进行加法运算 2 + 3 = 5，然后计算 5 * 4，最终得到 20。但如果你先进行乘法运算，3 * 4 = 12，接着 2 + 12，结果等于 14。与其他编程语言一样，在 Python 里，乘法的*优先级*要高于加法，因此第二种运算结果是正确的：

```
>>> 2 + 3 * 4
14
```

如何了解优先级规则？我在附录 F 中为你准备了一张优先级表，但在实际编程中我几乎从来没有查看过它，因为我们总可以使用括号来保证运算顺序与我们期望的一致：

```
>>> 2 + (3 * 4)
14
```

这样书写的代码也可让阅读者无需猜测代码的意图，免去了检查优先级表的麻烦。

2.2.3　基数

在 Python 中，整数默认使用十进制数（以 10 为底），除非你在数字前添加前缀，显式地指定使用其他基数（base）。也许你永远都不会在自己的代码中用到其他基数，但你很有可能在其他人编写的 Python 代码里见到它们。

我们大多数人都有 10 根手指 10 根脚趾（我家里倒是有只猫多了几根指头，但我从来没见过它用自己的指头数数）。因此，我们习惯这样计数：0，1，2，3，4，5，6，7，8，9。到了 9 之后，我们用光了所有的数字，于是将数字 1 放到“十位”，并把 0 放到“个位”。因此，10 代表“1 个十加 0 个一”。我们无法用一个字符代表数字“十”。接着是 11，12，一直到 19，然后仿照之前的做法，我们将新多出来的 1 加到十位来组成 20（2 个十加 0 个一），以此类推。

基数指的是在必须进位前可以使用的数字的最大数量。以 2 为底（二进制）时，可以使用的数字只有 0 和 1。这里的 0 和十进制的 0 代表的意义相同，1 和十进制的 1 所代表的意义也相同。然而以 2 为底时，1 与 1 相加得到的将是 10（1 个二加 0 个一）。

在 Python 中，除十进制外你还可以使用其他三种进制的数字：

- `0b` 或 `0B` 代表二进制（以 2 为底）
- `0o` 或 `0O` 代表八进制（以 8 为底）
- `0x` 或 `0X` 代表十六进制（以 16 为底）

Python 解释器会打印出它们对应的十进制整数。我们来试试这些不同进制的数。首先是单纯的十进制数字 `10`，代表“1 个十加 0 个一”。

```
>>> 10
10
```

接着，试试二进制（以 2 为底），代表“1（十进制）个二加上 0 个一”：

```
>>> 0b10
2
```

八进制（以 8 为底），代表“1（十进制）个八加上 0 个一”：

```
>>> 0o10
8
```

十六进制（以 16 为底），代表“1（十进制）个 16 加上 0 个一”：

```
>>> 0x10
16
```

可能你会好奇，十六进制用的是哪 16 个“数字”，它们是：0，1，2，3，4，5，6，7，8，9，a，b，c，d，e 以及 f。因此，`0xa` 代表十进制的 `10`，`0xf` 代表十进制的 `15`，`0xf` 加 `1` 等于 `0x10`（十进制 16）。

为什么要使用 10 以外的基数？因为它们在进行位运算时非常有用。有关位运算以及不同进制之间的转换将在第 7 章进行介绍。

2.2.4 类型转换

我们可以方便地使用 `int()` 函数将其他的 Python 数据类型转换为整型。它会保留传入数据的整数部分并舍去小数部分。

Python 里最简单的数据类型是布尔型，它只有两个可选值：`True` 和 `False`。当转换为整数时，它们分别代表 `1` 和 `0`：

```
>>> int(True)
1
>>> int(False)
0
```

当将浮点数转换为整数时，所有小数点后面的部分会被舍去：

```
>>> int(98.6)
98
>>> int(1.0e4)
10000
```

也可以将仅包含数字和正负号的字符串（如果你不知道字符串是什么，不用着急，先往后读，很快就会了解）转换为整数，下面有几个例子：

```
>>> int('99')
99
>>> int('-23')
-23
>>> int('+12')
12
```

将一个整数转换为整数没有太多意义，这既不会产生任何改变也不会造成任何损失：

```
>>> int(12345)
12345
```

如果你试图将一个与数字无关的类型转化为整数，会得到一个异常：

```
>>> int('99 bottles of beer on the wall')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '99 bottles of beer on the wall'
>>> int('')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: ''
```

尽管上面例子中的字符串的确是以有效数字（`99`）开头的，但它没有就此截止，后面的内容不是纯数字，无法被 `int()` 函数识别，因此抛出异常。

> 第 4 章会详细介绍异常。现在，你只需知道异常是 Python 处理程序错误的方式（不像有些语言不做处理直接造成程序崩溃）。在本书里，我会经常展示各种出现异常的情况，而不是假定程序总是正确运行的，这样你可以更加了解 Python 是如何处理程序错误的。

`int()` 可以接受浮点数或由数字组成的字符串，但无法接受包含小数点或指数的字符串：

```
>>> int('98.6')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '98.6'
>>> int('1.0e4')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '1.0e4'
```

如果混合使用多种不同的数字类型进行计算，Python 会自动地进行类型转换：

```
>>> 4 + 7.0
11.0
```

与整数或浮点数混合使用时，布尔型的 `False` 会被当作 `0` 或 `0.0`，`Ture` 会被当作 `1` 或 `1.0`：

```
>>> True + 2
3
>>> False + 5.0
5.0
```

2.2.5　一个int型有多大

在 Python 2 里，一个 `int` 型包含 32 位，可以存储从 −2 147 483 648 到 2 147 483 647 的整数。

一个 `long` 型会占用更多的空间：64 位，可以存储从 −9 223 372 036 854 775 808 到 9 223 372 036 854 775 807 的整数。

到了 Python 3，`long` 类型已不复存在，而 `int` 类型变为可以存储任意大小的整数，甚至超过 64 位。因此，你可以进行像下面一样计算（`10**100` 被赋值给名为 googol 的变量，这是 Google 最初的名字，但由于其拼写困难而被现在的名字所取代）：

```
>>>
>>> googol = 10**100
>>> googol
100000000000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000
>>> googol * googol
100000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
```

在许多其他编程语言中，进行类似上面的计算会造成**整数溢出**，这是因为计算中的数字或结果需要的存储空间超过了计算机所提供的（例如 32 位或 64 位）。在程序编写中，溢出会产生许多负面影响。而 Python 在处理超大数计算方面不会产生任何错误，这也是它的一

个加分点。

2.2.6　浮点数

整数全部由数字组成，而*浮点数*（在 Python 里称为 float）包含非数字的小数点。浮点数与整数很像：你可以使用运算符（`+`、`-`、`*`、`/`、`//`、`**` 和 `%`）以及 `divmod()` 函数进行计算。

使用 `float()` 函数可以将其他数字类型转换为浮点型。与之前一样，布尔型在计算中等价于 `1.0` 和 `0.0`：

```
>>> float(True)
1.0
>>> float(False)
0.0
```

将整数转换为浮点数仅仅需要添加一个小数点：

```
>>> float(98)
98.0
>>> float('99')
99.0
```

此外，也可以将包含有效浮点数（数字、正负号、小数点、指数及指数的前缀 `e`）的字符串转换为真正的浮点型数字：

```
>>> float('98.6')
98.6
>>> float('-1.5')
-1.5
>>> float('1.0e4')
10000.0
```

2.2.7　数学函数

Python 包含许多常用的数学函数，例如平方根、余弦函数，等等。这些内容被放到了附录 C，在那里我还会讨论 Python 的科学应用。

2.3　字符串

不是程序员的人经常会认为程序员一定都非常擅长数学，因为他们整天和数字打交道。但事实上，大多数程序员在处理字符串上花费的时间要远远超过处理数字的时间。逻辑思维（以及创造力）的重要性要远远超过数学能力。

对 Unicode 的支持使得 Python 3 可以包含世界上任何书面语言以及许多特殊符号。对于 Unicode 的支持是 Python 3 从 Python 2 分离出来的重要原因之一，也正是这一重要特性促使人们转向使用 Python 3。处理 Unicode 编码有时会非常困难，因此我将相关内容分散在了本书的不同地方。而本节只会使用 ASCII 编码的例子。

字符串型是我们学习的第一个 Python *序列*类型，它的本质是字符序列。

与其他语言不同的是，Python 字符串是不可变的。你无法对原字符串进行修改，但可以将字符串的一部分复制到新字符串，来达到相同的修改效果。很快你就会学到如何实现。

2.3.1 使用引号创建

将一系列字符包裹在一对单引号或一对双引号中即可创建字符串，就像下面这样：

```
>>> 'Snap'
'Snap'
>>> "Crackle"
'Crackle'
```

交互式解释器输出的字符串永远是用单引号包裹的，但无论使用哪种引号，Python 对字符串的处理方式都是一样的，没有任何区别。

既然如此，为什么要使用两种引号？这么做的好处是可以创建本身就包含引号的字符串，而不用使用转义符。可以在双引号包裹的字符串中使用单引号，或者在单引号包裹的字符串中使用双引号：

```
>>> "'Nay,' said the naysayer."
"'Nay,' said the naysayer."
>>> 'The rare double quote in captivity: ".'
'The rare double quote in captivity: ".'
>>> 'A "two by four" is actually 1 1/2" × 3 1/2".'
'A "two by four is" actually 1 1/2" × 3 1/2".'
>>> "'There's the man that shot my paw!' cried the limping hound."
"'There's the man that shot my paw!' cried the limping hound."
```

你还可以使用连续三个单引号 '''，或者三个双引号 """ 创建字符串：

```
>>> '''Boom!'''
'Boom'
>>> """Eek!"""
'Eek!'
```

三元引号在创建短字符串时没有什么特殊用处。它多用于创建*多行字符串*。下面的例子中，创建的字符串引用了 Edward Lear 的经典诗歌：

```
>>> poem =  '''There was a Young Lady of Norway,
... Who casually sat in a doorway;
... When the door squeezed her flat,
... She exclaimed, "What of that?"
... This courageous Young Lady of Norway.'''
>>>
```

（上面这段代码是在交互式解释器里输入的，第一行的提示符为 `>>>`，后面行的提示符为 `...`，直到再次输入三元引号暗示赋值语句的完结，此时光标跳转到下一行并再次以 `>>>` 提示输入。）

如果你尝试通过单独的单双引号创建多行字符串，在你完成第一行并按下回车时，Python 会弹出错误提示：

```
>>> poem = 'There was a young lady of Norway,
  File "<stdin>", line 1
    poem = 'There was a young lady of Norway,
                                            ^
SyntaxError: EOL while scanning string literal
>>>
```

在三元引号包裹的字符串中，每行的换行符以及行首或行末的空格都会被保留：

```
>>> poem2 = '''I do not like thee, Doctor Fell.
...     The reason why, I cannot tell.
...     But this I know, and know full well:
...     I do not like thee, Doctor Fell.
... '''
>>> print(poem2)
I do not like thee, Doctor Fell.
    The reason why, I cannot tell.
    But this I know, and know full well:
    I do not like thee, Doctor Fell.

>>>
```

值得注意的是，`print()` 函数的输出与交互式解释器的自动响应输出存在一些差异：

```
>>> poem2
'I do not like thee, Doctor Fell.\n    The reason why, I cannot tell.\n    But
this I know, and know full well:\n    I do not like thee, Doctor Fell.\n'
```

`print()` 会把包裹字符串的引号截去，仅输出其实际内容，易于阅读。它还会自动地在各个输出部分之间添加空格，并在所有输出的最后添加换行符：

```
>>> print(99, 'bottles', 'would be enough.')
99 bottles would be enough.
```

如果你不希望 `print()` 自动添加空格或换行，随后将学会如何避免它们。

解释器可以打印字符串以及像 `\n` 的*转义符*，关于转义符的内容你会在 2.3.3 节看到。

最后要指出的是 Python 允许*空串*的存在，它不包含任何字符且完全合法。你可以使用前面提到的任意一种方法创建一个空串：

```
>>> ''
''
>>> ""
''
>>> ''''''
''
>>> """"""
''
>>>
```

为什么会用到空字符串？有些时候你想要创建的字符串可能源自另一字符串的内容，这时需要先创建一个空白的模板，也就是一个空字符串。

```
>>> bottles = 99
>>> base = ''
```

```
>>> base += 'current inventory: '
>>> base += str(bottles)
>>> base
'current inventory: 99'
```

2.3.2　使用str()进行类型转换

使用 `str()` 可以将其他 Python 数据类型转换为字符串：

```
>>> str(98.6)
'98.6'
>>> str(1.0e4)
'10000.0'
>>> str(True)
'True'
```

当你调用 `print()` 函数或者进行*字符串差值*（string interpolation）时，Python 内部会自动使用 `str()` 将非字符串对象转换为字符串。第 7 章会了解到相关内容。

2.3.3　使用\转义

Python 允许你对某些字符进行*转义*操作，以此来实现一些难以单纯用字符描述的效果。在字符的前面添加反斜线符号 \ 会使该字符的意义发生改变。最常见的转义符是 `\n`，它代表换行符，便于你在一行内创建多行字符串。

```
>>> palindrome = 'A man,\nA plan,\nA canal:\nPanama.'
>>> print(palindrome)
A man,
A plan,
A canal:
Panama.
```

转义符 `\t`（tab 制表符）常用于对齐文本，之后会经常见到：

```
>>> print('\tabc')
    abc
>>> print('a\tbc')
a    bc
>>> print('ab\tc')
ab      c
>>> print('abc\t')
abc
```

（上面例子中，最后一个字符串的末尾包含了一个制表符，当然你无法在打印的结果中看到它。）

有时你可能还会用到 \' 和 \" 来表示单、双引号，尤其当该字符串由相同类型的引号包裹时：

```
>>> testimony = "\"I did nothing!\" he said. \"Not that either! Or the other
 thing.\""
>>> print(testimony)
"I did nothing!" he said. "Not that either! Or the other thing."
>>> fact = "The world's largest rubber duck was 54'2\" by 65'7\" by 105'"
```

```
>>> print(fact)
The world's largest rubber duck was 54'2" by 65'7" by 105'
```

如果你需要输出一个反斜线字符，连续输入两个反斜线即可：

```
>>> speech = 'Today we honor our friend, the backslash: \\.'
>>> print(speech)
Today we honor our friend, the backslash: \.
```

2.3.4 使用+拼接

在 Python 中，你可以使用 + 将多个字符串或字符串变量拼接起来，就像下面这样：

```
>>> 'Release the kraken! ' + 'At once!'
'Release the kraken! At once!'
```

也可以直接将一个*字面字符串*（非字符串变量）放到另一个的后面直接实现拼接：

```
>>> "My word! " "A gentleman caller!"
'My word! A gentleman caller!'
```

进行字符串拼接时，Python 并不会自动为你添加空格，需要显示定义。但当我们调用 `print()` 进行打印时，Python 会在各个参数之间自动添加空格并在结尾添加换行符：

```
>>> a = 'Duck.'
>>> b = a
>>> c = 'Grey Duck!'
>>> a + b + c
'Duck.Duck.Grey Duck!'
>>> print(a, b, c)
Duck. Duck. Grey Duck!
```

2.3.5 使用*复制

使用 * 可以进行字符串复制。试着把下面这几行输入到交互式解释器里，看看结果是什么：

```
>>> start = 'Na ' * 4 + '\n'
>>> middle = 'Hey ' * 3 + '\n'
>>> end = 'Goodbye.'
>>> print(start + start + middle + end)
```

2.3.6 使用[]提取字符

在字符串名后面添加 []，并在括号里指定*偏移量*可以提取该位置的单个字符。第一个字符（最左侧）的偏移量为 0，下一个是 1，以此类推。最后一个字符（最右侧）的偏移量也可以用 -1 表示，这样就不必从头数到尾。偏移量从右到左紧接着为 -2、-3，以此类推。

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
>>> letters[0]
'a'
>>> letters[1]
'b'
```

```
>>> letters[-1]
'z'
>>> letters[-2]
'y'
>>> letters[25]
'z'
>>> letters[5]
'f'
```

如果指定的偏移量超过了字符串的长度（记住，偏移量从 0 开始增加到字符串长度 −1），会得到一个异常提醒：

```
>>> letters[100]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: string index out of range
```

位置索引在其他序列类型（列表和元组）中的使用也是如此，你将在第 3 章见到。

由于字符串是不可变的，因此你无法直接插入字符或改变指定位置的字符。看看当我们试图将 'Henny' 改变为 'Penny' 时会发生什么：

```
>>> name = 'Henny'
>>> naume[0] = 'P'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

为了改变字符串，我们需要组合使用一些字符串函数，例如 replace()，以及分片操作（很快就会学到）：

```
>>> name = 'Henny'
>>> name.replace('H', 'P')
'Penny'
>>> 'P' + name[1:]
'Penny'
```

2.3.7　使用[start:end:step]分片

分片操作（slice）可以从一个字符串中抽取子字符串（字符串的一部分）。我们使用一对方括号、起始偏移量 start、终止偏移量 end 以及可选的步长 step 来定义一个分片。其中一些可以省略。分片得到的子串包含从 start 开始到 end 之前的全部字符。

- [:] 提取从开头到结尾的整个字符串
- [start:] 从 start 提取到结尾
- [:end] 从开头提取到 end - 1
- [start:end] 从 start 提取到 end - 1
- [start:end:step] 从 start 提取到 end - 1，每 step 个字符提取一个

与之前一样，偏移量从左至右从 0、1 开始，依次增加；从右至左从 −1、−2 开始，依次减小。如果省略 start，分片会默认使用偏移量 0（开头）；如果省略 end，分片会默认使用偏移量 −1（结尾）。

我们来创建一个由小写字母组成的字符串：

```
>>> letters = 'abcdefghijklmnopqrstuvwxyz'
```

仅仅使用 : 分片等价于使用 0 : -1（也就是提取整个字符串）：

```
>>> letters[:]
'abcdefghijklmnopqrstuvwxyz'
```

下面是一个从偏移量 20 提取到字符串结尾的例子：

```
>>> letters[20:]
'uvwxyz'
```

现在，从偏移量 10 提取到结尾：

```
>>> letters[10:]
'klmnopqrstuvwxyz'
```

下一个例子提取了偏移量从 12 到 14 的字符（Python 的提取操作不包含最后一个偏移量对应的字符）：

```
>>> letters[12:15]
'mno'
```

提取最后三个字符：

```
>>> letters[-3:]
'xyz'
```

下面一个例子提取了从偏移量为 18 的字符到倒数第 4 个字符。注意与上一个例子的区别：当偏移量 −3 作为开始位置时，将获得字符 x；而当它作为终止位置时，分片实际上会在偏移量 −4 处停止，也就是提取到字符 w：

```
>>> letters[18:-3]
'stuvw'
```

接下来，试着提取从倒数第 6 个字符到倒数第 3 个字符：

```
>>> letters[-6:-2]
'uvwx'
```

如果你需要的步长不是默认的 1，可以在第二个冒号后面进行指定，就像下面几个例子所示。

从开头提取到结尾，步长设为 7：

```
>>> letters[::7]
'ahov'
```

从偏移量 4 提取到偏移量 19，步长设为 3：

```
>>> letters[4:20:3]
'ehknqt'
```

从偏移量 19 提取到结尾，步长设为 4：

```
>>> letters[19::4]
'tx'
```

从开头提取到偏移量 20，步长设为 5：

```
>>> letters[:21:5]
'afkpu'
```

（记住，分片中 end 的偏移量需要比实际提取的最后一个字符的偏移量多 1。）

是不是非常方便？但这还没有完。如果指定的步长为负数，机智的 Python 还会从右到左反向进行提取操作。下面这个例子便从右到左以步长为 1 进行提取：

```
>>> letters[-1::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

事实上，你可以将上面的例子简化为下面这种形式，结果完全一致：

```
>>> letters[::-1]
'zyxwvutsrqponmlkjihgfedcba'
```

分片操作对于无效偏移量的容忍程度要远大于单字符提取操作。在分片中，小于起始位置的偏移量会被当作 `0`，大于终止位置的偏移量会被当作 `-1`，就像接下来几个例子展示的一样。

提取倒数 50 个字符：

```
>>> letters[-50:]
'abcdefghijklmnopqrstuvwxyz'
```

提取从倒数第 51 到倒数第 50 个字符：

```
>>> letters[-51:-50]
''
```

从开头提取到偏移量为 69 的字符：

```
>>> letters[:70]
'abcdefghijklmnopqrstuvwxyz'
```

从偏移量为 70 的字符提取到偏移量为 71 的字符：

```
>>> letters[70:71]
''
```

2.3.8 使用len()获得长度

到目前为止，我们已经学会了使用许多特殊的标点符号（例如 +）对字符串进行相应操作。但标点符号只有有限的几种。从现在开始，我们将学习使用 Python 的内置*函数*。所谓函数指的是可以执行某些特定操作的有名字的代码。

`len()` 函数可用于计算字符串包含的字符数：

```
>>> len(letters)
26
>>> empty = ""
```

```
>>> len(empty)
0
```

也可以对其他的序列类型使用 `len()`，这些内容都被放在第 3 章里讲解。

2.3.9 使用split()分割

与广义函数 `len()` 不同，有些函数只适用于字符串类型。为了调用字符串函数，你需要输入字符串的名称、一个点号，接着是需要调用的函数名，以及需要传入的参数：`string.function(arguments)`。4.7 节会学习更多关于函数的内容。

使用内置的字符串函数 `split()` 可以基于分隔符将字符串分割成由若干子串组成的列表。所谓列表（list）是由一系列值组成的序列，值与值之间由逗号隔开，整个列表被方括号所包裹。

```
>>> todos = 'get gloves,get mask,give cat vitamins,call ambulance'
>>> todos.split(',')
['get gloves', 'get mask', 'give cat vitamins', 'call ambulance']
```

上面例子中，字符串名为 `todos`，函数名为 `split()`，传入的参数为单一的分隔符 `','`。如果不指定分隔符，那么 `split()` 将默认使用空白字符——换行符、空格、制表符。

```
>>> todos.split()
['get', 'gloves,get', 'mask,give', 'cat', 'vitamins,call', 'ambulance']
```

即使不传入参数，调用 `split()` 函数时仍需要带着括号，这样 Python 才能知道你想要进行函数调用。

2.3.10 使用join()合并

可能你已经猜到了，`join()` 函数与 `split()` 函数正好相反：它将包含若干子串的列表分解，并将这些子串合成一个完整的大的字符串。`join()` 的调用顺序看起来有点别扭，与 `split()` 相反，你需要首先指定粘合用的字符串，然后再指定需要合并的列表：`string.join(list)`。因此，为了将列表 lines 中的多个子串合并成完整的字符串，我们应该使用语句：`'\n'.join(lines)`。下面的例子将列表中的名字通过逗号及空格粘合在一起：

```
>>> crypto_list = ['Yeti', 'Bigfoot', 'Loch Ness Monster']
>>> crypto_string = ', '.join(crypto_list)
>>> print('Found and signing book deals:', crypto_string)
Found and signing book deals: Yeti, Bigfoot, Loch Ness Monster
```

2.3.11 熟悉字符串

Python 拥有非常多的字符串函数。这一节将探索其中最常用的一些。我们的测试对象是下面的字符串，它源自纽卡斯尔伯爵 Margaret Cavendish 的不朽名篇 *What Is Liquid?*：

```
>>> poem = '''All that doth flow we cannot liquid name
Or else would fire and water be the same;
But that is liquid which is moist and wet
Fire that property can never get.
```

```
Then 'tis not cold that doth the fire put out
But 'tis the wet that makes it die, no doubt.'''
```

先做个小热身，试着提取开头的 13 个字符（偏移量为 0 到 12）：

```
>>> poem[:13]
'All that doth'
```

这首诗有多少个字符呢？（计入空格和换行符。）

```
>>> len(poem)
250
```

这首诗是不是以 `All` 开头呢？

```
>>> poem.startswith('All')
True
```

它是否以 `That's all, folks!?` 结尾？

```
>>> poem.endswith('That\'s all, folks!')
False
```

接下来，查一查诗中第一次出现单词 `the` 的位置（偏移量）：

```
>>> word = 'the'
>>> poem.find(word)
73
```

以及最后一次出现 `the` 的偏移量：

```
>>> poem.rfind(word)
214
```

`the` 在这首诗中出现了多少次？

```
>>> poem.count(word)
3
```

诗中出现的所有字符都是字母或数字吗？

```
>>> poem.isalnum()
False
```

并非如此，诗中还包括标点符号。

2.3.12 大小写与对齐方式

在这一节，我们将介绍一些不那么常用的字符串函数。我们的测试字符串如下所示：

```
>>> setup = 'a duck goes into a bar...'
```

将字符串收尾的 `.` 都删除掉：

```
>>> setup.strip('.')
'a duck goes into a bar'
```

由于字符串是不可变的，上面这些例子实际上没有一个对 setup 真正做了修改。它们都仅仅是获取了 setup 的值，进行某些操作后将操作结果赋值给了另一个新的字符串而已。

让字符串首字母变成大写：

```
>>> setup.capitalize()
'A duck goes into a bar...'
```

让所有单词的开头字母变成大写：

```
>>> setup.title()
'A Duck Goes Into A Bar...'
```

让所有字母都变成大写：

```
>>> setup.upper()
'A DUCK GOES INTO A BAR...'
```

将所有字母转换成小写：

```
>>> setup.lower()
'a duck goes into a bar...'
```

将所有字母的大小写转换：

```
>>> setup.swapcase()
'a DUCK GOES INTO A BAR...'
```

再来看看与格式排版相关的函数。这里，我们假设例子中的字符串被排版在指定长度（这里是 30 个字符）的空间里。

在 30 个字符位居中：

```
>>> setup.center(30)
'  a duck goes into a bar...   '
```

左对齐：

```
>>> setup.ljust(30)
'a duck goes into a bar...     '
```

右对齐：

```
>>> setup.rjust(30)
'     a duck goes into a bar...'
```

第 7 章更加详细地介绍字符串格式以及转换，包括如何使用 % 以及 format()。

2.3.13　使用replace()替换

使用 replace() 函数可以进行简单的子串替换。你需要传入的参数包括：需要被替换的子串，用于替换的新子串，以及需要替换多少处。最后一个参数如果省略则默认只替换第一

次出现的位置：

```
>>> setup.replace('duck', 'marmoset')
'a marmoset goes into a bar...'
```

修改最多 100 处：

```
>>> setup.replace('a ', 'a famous ', 100)
'a famous duck goes into a famous bar...'
```

当你准确地知道想要替换的子串是什么样子时，`replace()` 是个非常不错的选择。但使用时一定要小心！在上面第二个例子中，如果我们粗心地把需要替换的子串写成了单个字符的 `'a'` 而不是两个字符的 `'a '`（`a` 后面跟着一个空格）的话，会错误地将所有单词中出现的 `a` 也一并替换了：

```
>>> setup.replace('a', 'a famous', 100)
'a famous duck goes into a famous ba famousr...'
```

有时，你可能想确保被替换的子串是一个完整的词，或者某一个词的开头，等等。在这种情况下，你需要借助*正则表达式*，相关内容将在第 7 章进行介绍。

2.3.14 更多关于字符串的内容

Python 还内置了许多在此没有涉及的字符串函数，其中有一些你可以在之后的几章见到。除此之外，你可以在标准文档链接（https://docs.python.org/3/library/stdtypes.html#string-methods）获取所有字符串函数的细节。

2.4 练习

本章介绍了 Python 最基本的元素：数字、字符串以及变量。我们试着在交互式解释器里完成一些相关的练习。

(1) 一个小时有多少秒？这里，请把交互式解释器当作计算器使用，将每分钟的秒数（`60`）乘以每小时的分钟数（`60`）得到结果。

(2) 将上一个练习得到的结果（每小时的秒数）赋值给名为 `seconds_per_hour` 的变量。

(3) 一天有多少秒？用你的 `seconds_per_hour` 变量进行计算。

(4) 再次计算每天的秒数，但这一次将结果存储在名为 `seconds_per_day` 的变量中。

(5) 用 `seconds_per_day` 除以 `seconds_per_hour`，使用浮点除法（`/`）。

(6) 用 `seconds_per_day` 除以 `seconds_per_hour`，使用整数除法（`//`）。除了末尾的 `.0`，本练习所得结果是否与前一个练习用浮点数除法得到的结果一致？

第 3 章

Python容器：列表、元组、字典与集合

第 2 章介绍了 Python 最底层的基本数据类型：布尔型、整型、浮点型以及字符串型。如果把这些数据类型看作组成 Python 的原子，那么本章将要提到的数据结构就像分子一样。在这一章中，我们会把之前所学的基本 Python 类型以更为复杂的方式组织起来。这些数据结构以后会经常用到。在编程中，最常见的工作就是将数据进行拆分或合并，将其加工为特定的形式，而数据结构就是用以切分数据的钢锯以及合并数据的粘合枪。

3.1 列表和元组

大多数编程语言都有特定的数据结构来存储由一系列元素组成的序列，这些元素以它们所处的位置为索引：从第一个到最后一个依次编号。前一章已经见过 Python 字符串了，它本质上是字符组成的序列。你也在上一章初识了列表结构——由任意类型元素组成的序列。本章将更深入地了解列表。

除字符串外，Python 还有另外两种序列结构：元组和列表。它们都可以包含零个或多个元素。与字符串不同的是，元组和列表并不要求所含元素的种类相同，每个元素都可以是任何 Python 类型的对象。得益于此，你可以根据自己的需求和喜好创建具有任意深度及复杂度的数据结构。

为什么 Python 需要同时设定列表和元组这两种序列呢？这是因为元组是不可变的，当你给元组赋值时，这些值便被固定在了元组里，再也无法修改。然而，列表却是可变的，这意味着可以随意地插入或删除其中的元素。在后面的内容中，我会举许多关于这两种结构的例子，但重点会放在列表上。

说一点题外话，你可能听到过元组（tuple）的两种不同发音，究竟哪一种是正确的？是不是会担心，如果猜错了就会被别人认为是个装腔作势假装懂 Python 的人？其实你一点也不用担心。Python 之父吉多 · 范罗苏姆在 Twitter（https://twitter.com/gvanrossum/status/86144775731941376）上说：“每周的一三五我会把 tuple 念作 too-pull，而二四六我喜欢念作 tub-pull。至于礼拜天嘛，我从不会讨论这些。 :)”

3.2 列表

列表非常适合利用顺序和位置定位某一元素，尤其是当元素的顺序或内容经常发生改变时。与字符串不同，列表是可变的。你可以直接对原始列表进行修改：添加新元素、删除或覆盖已有元素。在列表中，具有相同值的元素允许出现多次。

3.2.1 使用[]或list()创建列表

列表可以由零个或多个元素组成，元素之间用逗号分开，整个列表被方括号所包裹：

```
>>> empty_list = [ ]
>>> weekdays = ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday']
>>> big_birds = ['emu', 'ostrich', 'cassowary']
>>> first_names = ['Graham', 'John', 'Terry', 'Terry', 'Michael']
```

也可以使用 `list()` 函数来创建一个空列表：

```
>>> another_empty_list = list()
>>> another_empty_list
[]
```

4.6 节会再介绍一种创建列表的方式，称为*列表推导*。

上面的例子中，只有 `weekdays` 列表充分利用了列表的顺序性。`first_names` 则展示了列表中的值允许重复这一性质。

如果你仅仅想要记录一些互不相同的值，而不在乎它们之间的顺序关系，集合（set）会是一个更好的选择。在上面的例子中，`big_birds` 就更适合存储在一个集合中。本章的后半部分会了解到一些关于集合的内容。

3.2.2 使用list()将其他数据类型转换成列表

Python 的 `list()` 函数可以将其他数据类型转换成列表类型。下面的例子将一个字符串转换成了由单个字母组成的列表：

```
>>> list('cat')
['c', 'a', 't']
```

接下来的例子将一个元组（在列表之后介绍）转换成了列表：

```
>>> a_tuple = ('ready', 'fire', 'aim')
>>> list(a_tuple)
['ready', 'fire', 'aim']
```

正如 2.3.9 节所述，使用 split() 可以依据分隔符将字符串切割成由若干子串组成的列表：

```
>>> birthday = '1/6/1952'
>>> birthday.split('/')
['1', '6', '1952']
```

如果待分割的字符串中包含连续的分隔符，那么在返回的列表中会出现空串元素：

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('/')
['a', 'b', '', 'c', 'd', '', '', 'e']
```

如果把上面例子中的分隔符改成 // 则会得到如下结果：

```
>>> splitme = 'a/b//c/d///e'
>>> splitme.split('//')
>>>
['a/b', 'c/d', '/e']
```

3.2.3　使用[offset]获取元素

和字符串一样，通过偏移量可以从列表中提取对应位置的元素：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[0]
'Groucho'
>>> marxes[1]
'Chico'
>>> marxes[2]
'Harpo'
```

同样，负偏移量代表从尾部开始计数：

```
>>> marxes[-1]
'Harpo'
>>> marxes[-2]
'Chico'
>>> marxes[-3]
'Groucho'
>>>
```

指定的偏移量对于待访问列表必须有效——该位置的元素在访问前已正确赋值。当指定的偏移量小于起始位置或者大于末尾位置时，会产生异常（错误）。下面的例子展示了访问第 6 个（偏移量为 5，从 0 开始计数）或者倒数第 5 个 Marx 兄弟时产生的异常：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range

>>> marxes[-5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

3.2.4　包含列表的列表

列表可以包含各种类型的元素，包括其他列表，如下所示：

```
>>> small_birds = ['hummingbird', 'finch']
>>> extinct_birds = ['dodo', 'passenger pigeon', 'Norwegian Blue']
>>> carol_birds = [3, 'French hens', 2, 'turtledoves']
>>> all_birds = [small_birds, extinct_birds, 'macaw', carol_birds]
```

all_birds 这个列表的结构是什么样子的？

```
>>> all_birds
[['hummingbird', 'finch'], ['dodo', 'passenger pigeon', 'Norwegian Blue'], 'macaw',
[3, 'French hens', 2, 'turtledoves']]
```

来访问第一个元素看看：

```
>>> all_birds[0]
['hummingbird', 'finch']
```

第一个元素还是一个列表：事实上，它就是 small_birds，也就是创建 all_birds 列表时设定的第一个元素。以此类推，不难猜测第二个元素是什么：

```
>>> all_birds[1]
['dodo', 'passenger pigeon', 'Norwegian Blue']
```

和预想的一样，这是之前指定的第二个元素 extinct_birds。如果想要访问 extinct_birds 的第一个元素，可以指定双重索引从 all_birds 中提取：

```
>>> all_birds[1][0]
'dodo'
```

上面例子中的 [1] 指向外层列表 all_birds 的第二个元素，而 [0] 则指向内层列表的第一个元素。

3.2.5　使用[offset]修改元素

就像可以通过偏移量访问某元素一样，你也可以通过赋值对它进行修改：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes[2] = 'Wanda'
>>> marxes
['Groucho', 'Chico', 'Wanda']
```

与之前一样，列表的偏移量必须是合法有效的。

通过这种方式无法修改字符串中的指定字符，因为字符串是不可变的。列表是可变的，因此你可以改变列表中的元素个数，以及元素的值。

3.2.6 指定范围并使用切片提取元素

你可以使用*切片*提取列表的一个子序列：

```
>>> marxes = ['Groucho', 'Chico,' 'Harpo']
>>> marxes[0:2]
['Groucho', 'Chico']
```

列表的切片仍然是一个列表。

与字符串一样，列表的切片也可以设定除 1 以外的步长。下面的例子从列表的开头开始每 2 个提取一个元素：

```
>>> marxes[::2]
['Groucho', 'Harpo']
```

再试试从尾部开始提取，步长仍为 2：

```
>>> marxes[::-2]
['Harpo', 'Groucho']
```

利用切片还可以巧妙地实现列表逆序：

```
>>> marxes[::-1]
['Harpo', 'Chico', 'Groucho']
```

3.2.7 使用append()添加元素至尾部

传统的向列表中添加元素的方法是利用 `append()` 函数将元素一个个添加到尾部。假设前面的例子中我们忘记了添加 Zeppo，没关系，由于列表是可变的，可以方便地把它添加到尾部：

```
>>> marxes.append('Zeppo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

3.2.8 使用extend()或+=合并列表

使用 `extend()` 可以将一个列表合并到另一个列表中。一个好心人又给了我们一份 Marx 兄弟的名字列表 `others`，我们希望能把它加到已有的 `marxes` 列表中：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.extend(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

也可以使用 +=：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes += others
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', 'Gummo', 'Karl']
```

如果错误地使用了 append()，那么 others 会被当成一个单独的元素进行添加，而不是将其中的内容进行合并：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> others = ['Gummo', 'Karl']
>>> marxes.append(others)
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo', ['Gummo', 'Karl']]
```

这个例子再次体现了列表可以包含不同类型的元素。上面的列表包含了四个字符串元素以及一个含有两个字符串的列表元素。

3.2.9　使用insert()在指定位置插入元素

append() 函数只能将新元素插入到列表尾部，而使用 insert() 可以将元素插入到列表的任意位置。指定偏移量为 0 可以插入列表头部。如果指定的偏移量超过了尾部，则会插入到列表最后，就如同 append() 一样，这一操作不会产生 Python 异常。

```
>>> marxes.insert(3, 'Gummo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes.insert(10, 'Karl')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo', 'Karl']
```

3.2.10　使用del删除指定位置的元素

校对员刚刚通知说 Gummo 确实是 Marx 兄弟的一员，但 Karl 并不是，因此我们需要撤销刚才最后插入的元素：

```
>>> del marxes[-1]
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
```

当列表中一个元素被删除后，位于它后面的元素会自动往前移动填补空出的位置，且列表长度减 1。再试试从更新后的 marxes 列表中删除 'Harpo'：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes[2]
'Harpo'
>>> del marxes[2]
>>> marxes
['Groucho', 'Chico', 'Gummo', 'Zeppo']
>>> marxes[2]
'Gummo'
```

del 是 Python 语句，而不是列表方法——无法通过 marxes[-2].del() 进行调用。del 就像是赋值语句（=）的逆过程：它将一个 Python 对象与它的名字分离。如果这个对象无其他名称引用，则其占用空间也被会清除。

3.2.11 使用remove()删除具有指定值的元素

如果不确定或不关心元素在列表中的位置，可以使用 remove() 根据指定的值删除元素。再见了，Gummo：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Gummo', 'Zeppo']
>>> marxes.remove('Gummo')
>>> marxes
['Groucho', 'Chico', 'Harpo', 'Zeppo']
```

3.2.12 使用pop()获取并删除指定位置的元素

使用 pop() 同样可以获取列表中指定位置的元素，但在获取完成后，该元素会被自动删除。如果你为 pop() 指定了偏移量，它会返回偏移量对应位置的元素；如果不指定，则默认使用 -1。因此，pop(0) 将返回列表的头元素，而 pop() 或 pop(-1) 则会返回列表的尾元素：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.pop()
'Zeppo'
>>> marxes
['Groucho', 'Chico', 'Harpo']
>>> marxes.pop(1)
'Chico'
>>> marxes
['Groucho', 'Harpo']
```

又到了计算机术语时间了！别担心，这些不在期末考试范围内。如果使用 append() 来添加元素到尾部，并通过 pop() 从尾部删除元素，实际上，实现了一个被称为 LIFO（后进先出）队列的数据结构。我们更习惯称之为栈（stack）。如果使用 pop(0) 来删除元素则创建了一个 FIFO（先进先出）队列。这两种数据结构非常有用，你可以不断接收数据，并根据需求对最先到达的数据（FIFO）或最后到达的数据（LIFO）进行处理。

3.2.13 使用index()查询具有特定值的元素位置

如果想知道等于某一个值的元素位于列表的什么位置，可以使用 index() 函数进行查询：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> marxes.index('Chico')
1
```

3.2.14 使用in判断值是否存在

判断一个值是否存在于给定的列表中有许多方式，其中最具有 Python 风格的是使用 in：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo', 'Zeppo']
>>> 'Groucho' in marxes
True
>>> 'Bob' in marxes
False
```

同一个值可能出现在列表的多个位置，但只要至少出现一次，in 就会返回 True：

```
>>> words = ['a', 'deer', 'a' 'female', 'deer']
>>> 'deer' in words
True
```

如果经常需要判断一个值是否存在于一个列表中，但并不关心列表中元素之间的顺序，那么使用 Python 集合进行存储和查找会是更好的选择。本章的稍后部分会介绍一点关于集合的内容。

3.2.15 使用count()记录特定值出现的次数

使用 count() 可以记录某一个特定值在列表中出现的次数：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> marxes.count('Harpo')
1
>>> marxes.count('Bob')
0

>>> snl_skit = ['cheeseburger', 'cheeseburger', 'cheeseburger']
>>> snl_skit.count('cheeseburger')
3
```

3.2.16 使用join()转换为字符串

2.3.10 节详细地讨论了 join() 的使用方法，这里又提供了一种新的使用方式：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> ', '.join(marxes)
'Groucho, Chico, Harpo'
```

等等，你可能会觉得 join() 的使用顺序看起来有点别扭。这是因为 join() 实际上是一个字符串方法，而不是列表方法。不能通过 marxes.join(',') 进行调用，尽管这可能看起来更直观。join() 函数的参数是字符串或者其他可迭代的包含字符串的序列（例如上面例子中的字符串列表），它的输出是一个字符串。如果 join() 是列表方法，将无法对其他可迭代的对象（例如元组、字符串）使用。如果坚持想让它能接受任何迭代类型，你必须亲自为每一种类型的序列编写合并代码，这太过费力。试着这样来记忆 join() 的调用顺序：join() 是 split() 的逆过程，如下所示：

```
>>> friends = ['Harry', 'Hermione', 'Ron']
>>> separator = ' * '
>>> joined = separator.join(friends)
>>> joined
'Harry * Hermione * Ron'
```

```
>>> separated = joined.split(separator)
>>> separated
['Harry', 'Hermione', 'Ron']
>>> separated == friends
True
```

3.2.17　使用sort()重新排列元素

在实际应用中，经常需要将列表中的元素按值排序，而不是按照偏移量排序。Python 为此提供了两个函数：

- 列表方法 sort() 会对原列表进行排序，改变原列表内容；
- 通用函数 sorted() 则会返回排好序的列表副本，原列表内容不变。

如果列表中的元素都是数字，它们会默认地被排列成从小到大的升序。如果元素都是字符串，则会按照字母表顺序排列：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> sorted_marxes = sorted(marxes)
>>> sorted_marxes
['Chico', 'Groucho', 'Harpo']
```

sorted_marxes 是一个副本，它的创建并不会改变原始列表的内容：

```
>>> marxes
['Groucho', 'Chico', 'Harpo']
```

但对 marxes 列表调用列表函数 sort() 则会改变它的内容：

```
>>> marxes.sort()
>>> marxes
['Chico', 'Groucho', 'Harpo']
```

当列表中的所有元素都是同一种类型时（例如 marxes 中都是字符串），sort() 会正常工作。有些时候甚至多种类型也可——例如整型和浮点型——只要它们之间能够自动地互相转换：

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort()
>>> numbers
[1, 2, 3, 4.0]
```

默认的排序是升序的，通过添加参数 reverse=True 可以改变为降序排列：

```
>>> numbers = [2, 1, 4.0, 3]
>>> numbers.sort(reverse=True)
>>> numbers
[4.0, 3, 2, 1]
```

3.2.18　使用len()获取长度

len() 可以返回列表长度：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> len(marxes)
3
```

3.2.19 使用=赋值，使用copy()复制

如果将一个列表赋值给了多个变量，改变其中的任何一处会造成其他变量对应的值也被修改，如下所示：

```
>>> a = [1, 2, 3]
>>> a
[1, 2, 3]
>>> b = a
>>> b
[1, 2, 3]
>>> a[0] = 'surprise'
>>> a
['surprise', 2, 3]
```

现在，b 的值是什么？它会保持 [1, 2, 3]，还是改变为 ['surprise', 2, 3]？试一试：

```
>>> b
['surprise', 2, 3]
```

还记得第 2 章中贴标签的比喻吗？ b 与 a 实际上指向的是同一个对象，因此，无论我们是通过 a 还是通过 b 来修改列表的内容，其结果都会作用于双方：

```
>>> b
['surprise', 2, 3]
>>> b[0] = 'I hate surprises'
>>> b
['I hate surprises', 2, 3]
>>> a
['I hate surprises', 2, 3]
```

通过下面任意一种方法，都可以将一个列表的值复制到另一个新的列表中：

- 列表 copy() 函数
- list() 转换函数
- 列表分片 [:]

测试初始时我们的列表叫作 a，然后利用 copy() 函数创建 b，利用 list() 函数创建 c，并使用列表分片创建 d：

```
>>> a = [1, 2, 3]
>>> b = a.copy()
>>> c = list(a)
>>> d = a[:]
```

再次注意，在这个例子中，b、c、d 都是 a 的复制：它们是自身带有值的新对象，与原始的 a 所指向的列表对象 [1, 2, 3] 没有任何关联。改变 a 不影响 b、c 和 d 的复制：

```
>>> a[0] = 'integer lists are boring'
>>> a
```

```
['integer lists are boring', 2, 3]
>>> b
[1, 2, 3]
>>> c
[1, 2, 3]
>>> d
[1, 2, 3]
```

3.3 元组

与列表类似，元组也是由任意类型元素组成的序列。与列表不同的是，元组是不可变的，这意味着一旦元组被定义，将无法再进行增加、删除或修改元素等操作。因此，元组就像是一个常量列表。

3.3.1 使用()创建元组

下面的例子展示了创建元组的过程，它的语法与我们直观上预想的有一些差别。

可以用 () 创建一个空元组：

```
>>> empty_tuple = ()
>>> empty_tuple
()
```

创建包含一个或多个元素的元组时，每一个元素后面都需要跟着一个逗号，即使只包含一个元素也不能省略：

```
>>> one_marx = 'Groucho',
>>> one_marx
('Groucho',)
```

如果创建的元组所包含的元素数量超过 1，最后一个元素后面的逗号可以省略：

```
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

Python 的交互式解释器输出元组时会自动添加一对圆括号。你并不需要这么做——定义元组真正靠的是每个元素的后缀逗号——但如果你习惯添加一对括号也无可厚非。可以用括号将所有元素包裹起来，这会使得程序更加清晰：

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> marx_tuple
('Groucho', 'Chico', 'Harpo')
```

可以一口气将元组赋值给多个变量：

```
>>> marx_tuple = ('Groucho', 'Chico', 'Harpo')
>>> a, b, c = marx_tuple
>>> a
'Groucho'
>>> b
```

```
'Chico'
>>> c
'Harpo'
```

有时这个过程被称为*元组解包*。

可以利用元组在一条语句中对多个变量的值进行交换，而不需要借助临时变量：

```
>>> password = 'swordfish'
>>> icecream = 'tuttifrutti'
>>> password, icecream = icecream, password
>>> password
'tuttifrutti'
>>> icecream
'swordfish'
>>>
```

`tuple()` 函数可以用其他类型的数据来创建元组：

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> tuple(marx_list)
('Groucho', 'Chico', 'Harpo')
```

3.3.2　元组与列表

在许多地方都可以用元组代替列表，但元组的方法函数与列表相比要少一些——元组没有 `append()`、`insert()`，等等——因为一旦创建元组便无法修改。既然列表更加灵活，那为什么不在所有地方都使用列表呢？原因如下所示：

- 元组占用的空间较小
- 你不会意外修改元组的值
- 可以将元组用作字典的键（详见 3.4 节）
- *命名元组*（详见第 6 章“命名元组”小节）可以作为对象的替代
- 函数的参数是以元组形式传递的（详见 4.7 节）

这一节不会再介绍更多关于元组的细节了。实际编程中，更多场合用到的是列表和字典，而接下来要介绍的就是字典结构。

3.4　字典

字典（dictionary）与列表类似，但其中元素的顺序无关紧要，因为它们不是通过像 0 或 1 的偏移量访问的。取而代之，每个元素拥有与之对应的互不相同的*键*（key），需要通过键来访问元素。键通常是字符串，但它还可以是 Python 中其他任意的不可变类型：布尔型、整型、浮点型、元组、字符串，以及其他一些在后面的内容中会见到的类型。字典是可变的，因此你可以增加、删除或修改其中的键值对。

如果使用过只支持数组或列表的语言，那么你很快就会爱上 Python 里的字典类型。

在其他语言中，字典可能会被称作关系型数组、哈希表或哈希图。在 Python 中，字典（dictionary）还经常会被简写成 dict。

3.4.1 使用{}创建字典

用大括号（{}）将一系列以逗号隔开的键值对（key:value）包裹起来即可进行字典的创建。最简单的字典是空字典，它不包含任何键值对：

```
>>> empty_dict = {}
>>> empty_dict
{}
```

引用 Ambrose Bierce 的《魔鬼词典》（*The Devil's Dictionary*）来创建一个字典：

```
>>> bierce = {
...     "day": "A period of twenty-four hours, mostly misspent",
...     "positive": "Mistaken at the top of one's voice",
...     "misfortune": "The kind of fortune that never misses",
...     }
>>>
```

在交互式解释器中输入字典名会打印出它所包含的所有键值对：

```
>>> bierce
{'misfortune': 'The kind of fortune that never misses',
'positive': "Mistaken at the top of one's voice",
'day': 'A period of twenty-four hours, mostly misspent'}
```

Python 允许在列表、元组或字典的最后一个元素后面添加逗号，这不会产生任何问题。此外，在括号之间输入键值对来创建字典时并不强制缩进，我这么做只是为了增加代码的可读性。

3.4.2 使用dict()转换为字典

可以用 dict() 将包含双值子序列的序列转换成字典。（你可能会经常遇到这种子序列，例如“Strontium，90，Carbon，14”或者“Vikings，20，Packers，7”，等等。）每个子序列的第一个元素作为键，第二个元素作为值。

首先，这里有一个使用 lol（a list of two-item list）创建字典的小例子：

```
>>> lol = [ ['a', 'b'], ['c', 'd'], ['e', 'f'] ]
>>> dict(lol)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

记住，字典中元素的顺序是无关紧要的，实际存储顺序可能取决于你添加元素的顺序。

可以对任何包含双值子序列的序列使用 `dict()`，下面是其他例子。

包含双值元组的列表：

```
>>> lot = [ ('a', 'b'), ('c', 'd'), ('e', 'f') ]
>>> dict(lot)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

包含双值列表的元组：

```
>>> tol = ( ['a', 'b'], ['c', 'd'], ['e', 'f'] )
>>> dict(tol)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

双字符的字符串组成的列表：

```
>>> los = [ 'ab', 'cd', 'ef' ]
>>> dict(los)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

双字符的字符串组成的元组：

```
>>> tos = ( 'ab', 'cd', 'ef' )
>>> dict(tos)
{'c': 'd', 'a': 'b', 'e': 'f'}
```

4.5.4 节会教你如何使用 `zip()` 函数，利用它能非常简单地创建上面这种双元素序列。

3.4.3　使用[key]添加或修改元素

向字典中添加元素非常简单，只需指定该元素的键并赋予相应的值即可。如果该元素的键已经存在于字典中，那么该键对应的旧值会被新值取代。如果该元素的键并未在字典中出现，则会被加入字典。与列表不同，你不需要担心赋值过程中 Python 会抛出越界异常。

我们来建立一个包含大多数 Monty Python 成员名字的字典，用他们的姓当作键，名当作值：

```
>>> pythons = {
...     'Chapman': 'Graham',
...     'Cleese': 'John',
...     'Idle': 'Eric',
...     'Jones': 'Terry',
...     'Palin': 'Michael',
...     }
>>> pythons
{'Cleese': 'John', 'Jones': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric'}
```

等等，我们好像落下了一个人：生于美国的 Terry Gilliam。下面是一个糟糕的程序员为了将 Gilliam 添加进字典而编写的代码，他粗心地将 Gilliam 的名字打错了：

```
>>> pythons['Gilliam'] = 'Gerry'
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Gerry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

下面是另一位颇具 Python 风格的程序员的修补代码：

```
>>> pythons['Gilliam'] = 'Terry'
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

通过使用相同的键（`'Gilliam'`）将原本的对应值 `'Gerry'` 修改为了 `'Terry'`。

记住，字典的键必须保证互不相同。这就是为什么在这里使用姓作为键，而不是使用名——Monty Python 的成员中有两个都叫 Terry！如果创建字典时同一个键出现了两次，那么后面出现的值会取代之前的值：

```
>>> some_pythons = {
...     'Graham': 'Chapman',
...     'John': 'Cleese',
...     'Eric': 'Idle',
...     'Terry': 'Gilliam',
...     'Michael': 'Palin',
...     'Terry': 'Jones',
...     }
>>> some_pythons
{'Terry': 'Jones', 'Eric': 'Idle', 'Graham': 'Chapman',
'John': 'Cleese', 'Michael': 'Palin'}
```

上面例子中，我们首先将 `'Gilliam'` 赋值给了键 `'Terry'`，然后又用 `'Jones'` 把它取代掉了。

3.4.4 使用update()合并字典

使用 `update()` 可以将一个字典的键值对复制到另一个字典中去。

首先定义一个包含所有成员的字典 `pythons`：

```
>>> pythons = {
...     'Chapman': 'Graham',
...     'Cleese': 'John',
...     'Gilliam': 'Terry',
...     'Idle': 'Eric',
...     'Jones': 'Terry',
...     'Palin': 'Michael',
...     }
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

接着定义一个包含其他喜剧演员的字典，命名为 `others`：

```
>>> others = { 'Marx': 'Groucho', 'Howard': 'Moe' }
```

现在，出现了另一个糟糕的程序员，它认为 `others` 应该被归入 Monty Python 成员中：

```
>>> pythons.update(others)
>>> pythons
{'Cleese': 'John', 'Howard': 'Moe', 'Gilliam': 'Terry',
'Palin': 'Michael', 'Marx': 'Groucho', 'Chapman': 'Graham',
'Idle': 'Eric', 'Jones': 'Terry'}
```

如果待添加的字典与待扩充的字典包含同样的键会怎样？是的，新归入字典的值会取代原有的值：

```
>>> first = {'a': 1, 'b': 2}
>>> second = {'b': 'platypus'}
>>> first.update(second)
>>> first
{'b': 'platypus', 'a': 1}
```

3.4.5 使用del删除具有指定键的元素

技术上来说，上面那个糟糕的程序员写的代码倒是正确的。但是他不应该这么做！others 里的成员虽然也很搞笑很出名，但他们终归不是 Monty Python 的成员。把最后添加的两个成员清除出去：

```
>>> del pythons['Marx']
>>> pythons
{'Cleese': 'John', 'Howard': 'Moe', 'Gilliam': 'Terry',
'Palin': 'Michael', 'Chapman': 'Graham', 'Idle': 'Eric',
'Jones': 'Terry'}
>>> del pythons['Howard']
>>> pythons
{'Cleese': 'John', 'Gilliam': 'Terry', 'Palin': 'Michael',
'Chapman': 'Graham', 'Idle': 'Eric', 'Jones': 'Terry'}
```

3.4.6 使用clear()删除所有元素

使用 clear()，或者给字典变量重新赋值一个空字典（{}）可以将字典中所有元素删除：

```
>>> pythons.clear()
>>> pythons
{}
>>> pythons = {}
>>> pythons
{}
```

3.4.7 使用in判断是否存在

如果你希望判断某一个键是否存在于一个字典中，可以使用 in。我们来重新定义一下 pythons 字典，这一次可以省略一两个人名：

```
>>> pythons = {'Chapman': 'Graham', 'Cleese': 'John',
'Jones': 'Terry', 'Palin': 'Michael'}
```

测试一下看看谁在里面：

```
>>> 'Chapman' in pythons
True
>>> 'Palin' in pythons
True
```

这一次记得把 Terry Gilliam 添加到成员字典里了吗？

```
>>> 'Gilliam' in pythons
False
```

糟糕，好像又忘记了。

3.4.8　使用[key]获取元素

这是对字典最常进行的操作，只需指定字典名和键即可获得对应的值：

```
>>> pythons['Cleese']
'John'
```

如果字典中不包含指定的键，会产生一个异常：

```
>>> pythons['Marx']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'Marx'
```

有两种方法可以避免这种情况的发生。第一种是在访问前通过 in 测试键是否存在，就像在上一小节看到的一样：

```
>>> 'Marx' in pythons
False
```

另一种方法是使用字典函数 get()。你需要指定字典名，键以及一个可选值。如果键存在，会得到与之对应的值：

```
>>> pythons.get('Cleese')
'John'
```

反之，若键不存在，如果你指定了可选值，那么 get() 函数将返回这个可选值：

```
>>> pythons.get('Marx', 'Not a Python')
'Not a Python'
```

否则，会得到 None（在交互式解释器中什么也不会显示）：

```
>>> pythons.get('Marx')
>>>
```

3.4.9　使用keys()获取所有键

使用 keys() 可以获得字典中的所有键。在接下来的几个例子中，我们将换一个示例：

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> signals.keys()
dict_keys(['green', 'red', 'yellow'])
```

在 Python 2 里，keys() 会返回一个列表，而在 Python 3 中则会返回 dict_keys()，它是键的迭代形式。这种返回形式对于大型的字典非常有用，因为它不需要时间和空间来创建返回的列表。有时你需要的可能就是一个完整的列表，但在 Python 3 中，你只能自己调用 list() 将 dict_keys 转换为列表类型。

```
>>> list( signals.keys() )
['green', 'red', 'yellow']
```

在 Python 3 里，你同样需要手动使用 list() 将 values() 和 items() 的返回值转换为普通的 Python 列表。之后的例子中会用到这些。

3.4.10 使用values()获取所有值

使用 values() 可以获取字典中的所有值：

```
>>> list( signals.values() )
['go', 'smile for the camera', 'go faster']
```

3.4.11 使用items()获取所有键值对

使用 items() 函数可以获取字典中所有的键值对：

```
>>> list( signals.items() )
[('green', 'go'), ('red', 'smile for the camera'), ('yellow', 'go faster')]
```

每一个键值对以元组的形式返回，例如 ('green','go')。

3.4.12 使用=赋值，使用copy()复制

与列表一样，对字典内容进行的修改会反映到所有与之相关联的变量名上：

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> save_signals = signals
>>> signals['blue'] = 'confuse everyone'
>>> save_signals
{'blue': 'confuse everyone', 'green': 'go',
'red': 'smile for the camera', 'yellow': 'go faster'}
```

若想避免这种情况，可以使用 copy() 将字典复制到一个新的字典中：

```
>>> signals = {'green': 'go', 'yellow': 'go faster', 'red': 'smile for the camera'}
>>> original_signals = signals.copy()
>>> signals['blue'] = 'confuse everyone'
>>> signals
{'blue': 'confuse everyone', 'green': 'go',
'red': 'smile for the camera', 'yellow': 'go faster'}
>>> original_signals
{'green': 'go', 'red': 'smile for the camera', 'yellow': 'go faster'}
```

3.5 集合

集合就像舍弃了值，仅剩下键的字典一样。键与键之间也不允许重复。如果你仅仅想知道某一个元素是否存在而不关心其他的，使用集合是个非常好的选择。如果需要为键附加其他信息的话，建议使用字典。

很久以前，当你还在小学时，可能就学到过一些关于集合论的知识。当然，如果你的学校恰

好跳过了这一部分内容（或者实际上教了，但你当时正好盯着窗外发呆，我小时候就总是这样开小差），可以仔细看看图 3-1，它展示了我们对于集合进行的最基本的操作——交和并。

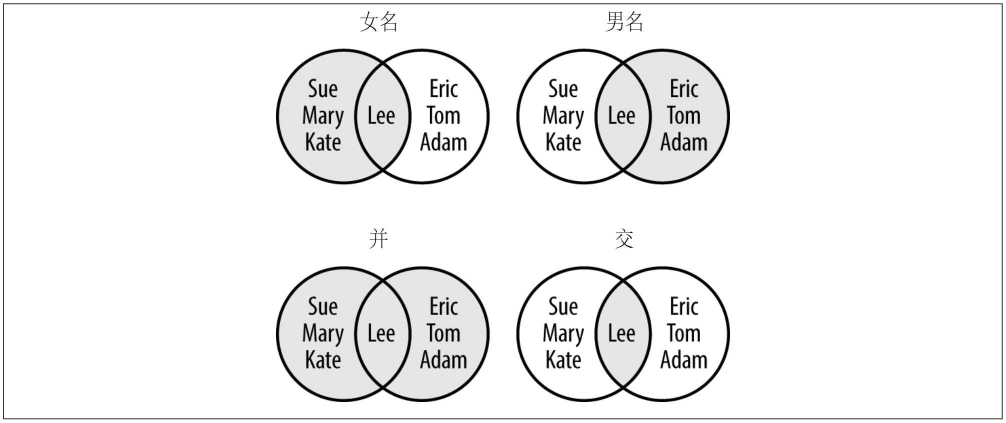


图 3-1：集合的常见操作

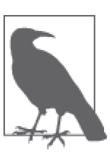
假如你将两个包含相同键的集合进行并操作，由于集合中的元素只能出现一次，因此得到的并集将包含两个集合所有的键，但每个键仅出现一次。空或空集指的是包含零个元素的集合。在图 3-1 中，名字以 X 开头的女性组成的集合就是一个空集。

3.5.1　使用set()创建集合

你可以使用 set() 函数创建一个集合，或者用大括号将一系列以逗号隔开的值包裹起来，如下所示：

```
>>> empty_set = set()
>>> empty_set
set()
>>> even_numbers = {0, 2, 4, 6, 8}
>>> even_numbers
{0, 8, 2, 4, 6}
>>> odd_numbers = {1, 3, 5, 7, 9}
>>> odd_numbers
{9, 3, 1, 5, 7}
```

与字典的键一样，集合是无序的。



由于 [] 能创建一个空列表，你可能期望 {} 也能创建空集。但事实上，{} 会创建一个空字典，这也是为什么交互式解释器把空集输出为 set() 而不是 {}。为何如此？没有什么特殊原因，仅仅是因为字典出现的比较早并抢先占据了花括号。

3.5.2　使用set()将其他类型转换为集合

你可以利用已有列表、字符串、元组或字典的内容来创建集合，其中重复的值会被丢弃。

首先来试着转换一个包含重复字母的字符串：

```
>>> set( 'letters' )
{'l', 'e', 't', 'r', 's'}
```

注意，上面得到的集合中仅含有一个 'e' 和一个 't'，尽管字符串 'letters' 里各自包含两个。

再试试用列表建立集合：

```
>>> set( ['Dasher', 'Dancer', 'Prancer', 'Mason-Dixon'] )
{'Dancer', 'Dasher', 'Prancer', 'Mason-Dixon'}
```

再试试元组：

```
>>> set( ('Ummagumma', 'Echoes', 'Atom Heart Mother') )
{'Ummagumma', 'Atom Heart Mother', 'Echoes'}
```

当字典作为参数传入 set() 函数时，只有键会被使用：

```
>>> set( {'apple': 'red', 'orange': 'orange', 'cherry': 'red'} )
{'apple', 'cherry', 'orange'}
```

3.5.3　使用in测试值是否存在

这是集合里最常用的功能。我们来建立一个叫 drinks 的字典。每个键都是一种混合饮料的名字，与之对应的值是配料组成的集合：


```
>>> drinks = {
...     'martini': {'vodka', 'vermouth'},
...     'black russian': {'vodka', 'kahlua'},
...     'white russian': {'cream', 'kahlua', 'vodka'},
...     'manhattan': {'rye', 'vermouth', 'bitters'},
...     'screwdriver': {'orange juice', 'vodka'}
...     }
```

尽管都由花括号（{ 和 }）包裹，集合仅仅是一系列值组成的序列，而字典是一个或多个键值对组成的序列。

哪种饮料含有伏特加？（注意，后面例子中我会提前使用一点下一章出现的 for、if、and 以及 or 语句。）

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents:
...         print(name)
...
screwdriver
martini
black russian
white russian
```

我想挑的饮料需要有伏特加，但不含乳糖。此外，我很讨厌苦艾酒，觉得它尝起来就像煤油一样：

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents and not ('vermouth' in contents or
...         'cream' in contents):
...         print(name)
...
screwdriver
black russian
```

后面的内容会将上面的代码改写得更加简洁。

3.5.4　合并及运算符

如果想要查看多个集合之间组合的结果应该怎么办？例如，你想要找到一种饮料，它含有果汁或含有苦艾酒。我们可以使用交集运算符，记作 &：

```
>>> for name, contents in drinks.items():
...     if contents & {'vermouth', 'orange juice'}:
...         print(name)
...
screwdriver
martini
manhattan
```

& 运算符的结果是一个集合，它包含所有同时出现在你比较的两个清单中的元素。在上面代码中，如果 contents 里面不包含任何一种指定成分，则 & 会返回一个空集，相当于 False。

现在来改写一下上一小节的例子，就是那个我们想要伏特加但不需要乳脂也不需要苦艾酒的例子：

```
>>> for name, contents in drinks.items():
...     if 'vodka' in contents and not contents & {'vermouth', 'cream'}:
...         print(name)
...
screwdriver
black russian
```

将这两种饮料的原料都存储到变量中，以便于后面的例子不用再重复输入：

```
>>> bruss = drinks['black russian']
>>> wruss = drinks['white russian']
```

之后的例子会涵盖所有的集合运算符。有些运算使用特殊定义过的标点，另一些则使用函数，还有一些运算两者都可使用。这里使用测试集合 a（包含 1 和 2），以及 b（包含 2 和 3）：

```
>>> a = {1, 2}
>>> b = {2, 3}
```

可以通过使用特殊标点符号 & 或者集合函数 intersection() 获取集合的交集（两集合共有元素），如下所示：

```
>>> a & b
{2}
>>> a.intersection(b)
{2}
```

下面的代码片段使用了我们之前存储的饮料变量：

```
>>> bruss & wruss
{'kahlua', 'vodka'}
```

下面的例子中，使用 | 或者 union() 函数来获取集合的*并集*（至少出现在一个集合中的元素）：

```
>>> a | b
{1, 2, 3}
>>> a.union(b)
{1, 2, 3}
```

还有酒精饮料的例子：

```
>>> bruss | wruss
{'cream', 'kahlua', 'vodka'}
```

使用字符 - 或者 difference() 可以获得两个集合的*差集*（出现在第一个集合但不出现在第二个集合）：

```
>>> a - b
{1}
>>> a.difference(b)
{1}

>>> bruss - wruss
set()
>>> wruss - bruss
{'cream'}
```

到目前为止，出现的都是最常见的集合运算，包括交、并、差。出于完整性的考虑，我会在接下来的例子中列出其他的运算符，这些运算符并不常用。

使用 ^ 或者 symmetric_difference() 可以获得两个集合的*异或集*（仅在两个集合中出现一次）：

```
>>> a ^ b
{1, 3}
>>> a.symmetric_difference(b)
{1, 3}
```

下面的代码帮我们找到了两种俄罗斯饮料的不同成分：

```
>>> bruss ^ wruss
{'cream'}
```

使用 <= 或者 issubset() 可以判断一个集合是否是另一个集合的*子集*（第一个集合的所有元素都出现在第二个集合中）：

```
>>> a <= b
False
>>> a.issubset(b)
False
```

向“黑俄罗斯酒”中加入一些乳脂就变成了“白俄罗斯酒”，因此 wruss 是 bruss 的超集：

```
>>> bruss <= wruss
True
```

一个集合是它本身的子集吗？答案为：是的。

```
>>> a <= a
True
>>> a.issubset(a)
True
```

当第二个集合包含所有第一个集合的元素，且仍包含其他元素时，我们称第一个集合为第二个集合的*真子集*。使用 < 可以进行判断：

```
>>> a < b
False
>>> a < a
False

>>> bruss < wruss
True
```

*超集*与子集正好相反（第二个集合的所有元素都出现在第一个集合中），使用 >= 或者 issuperset() 可以进行判断：

```
>>> a >= b
False
>>> a.issuperset(b)
False

>>> wruss >= bruss
True
```

一个集合是它本身的超集：

```
>>> a >= a
True
>>> a.issuperset(a)
True
```

最后，使用 > 可以找到一个集合的*真超集*（第一个集合包含第二个集合的所有元素且还包含其他元素）：

```
>>> a > b
False

>>> wruss > bruss
True
```

一个集合并不是它本身的真超集：

```
>>> a > a
False
```

3.6 比较几种数据结构

回顾一下，我们学会了使用方括号（[]）创建列表，使用逗号创建元组，使用花括号（{}）创建字典。在每一种类型中，都可以通过方括号对单个元素进行访问：

```
>>> marx_list = ['Groucho', 'Chico', 'Harpo']
>>> marx_tuple = 'Groucho', 'Chico', 'Harpo'
>>> marx_dict = {'Groucho': 'banjo', 'Chico': 'piano', 'Harpo': 'harp'}
>>> marx_list[2]
'Harpo'
>>> marx_tuple[2]
'Harpo'
>>> marx_dict['Harpo']
'harp'
```

对于列表和元组来说，方括号里的内容是整型的偏移量；而对于字典来说，方括号里的是键。它们返回的都是元素的值。

3.7 建立大型数据结构

我们从最简单的布尔型、数字、字符串型开始学习，到目前为止，学习到了列表、元组、集合以及字典等数据结构。你可以将这些内置的数据结构自由地组合成更大、更复杂的结构。试着从建立三个不同的列表开始：

```
>>> marxes = ['Groucho', 'Chico', 'Harpo']
>>> pythons = ['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin']
>>> stooges = ['Moe', 'Curly', 'Larry']
```

可以把上面每一个列表当作一个元素，并建立一个元组：

```
>>> tuple_of_lists = marxes, pythons, stooges
>>> tuple_of_lists
(['Groucho', 'Chico', 'Harpo'],
['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin'],
['Moe', 'Curly', 'Larry'])
```

同样，可以创建一个包含上面三个列表的列表：

```
>>> list_of_lists = [marxes, pythons, stooges]
>>> list_of_lists
[['Groucho', 'Chico', 'Harpo'],
['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin'],
['Moe', 'Curly', 'Larry']]
```

还可以创建以这三个列表为值的字典。本例中，使用这些喜剧演员组合的名字作为键，成员列表作为值：

```
>>> dict_of_lists = {'Marxes': marxes, 'Pythons': pythons, 'Stooges': stooges}
>> dict_of_lists
{'Stooges': ['Moe', 'Curly', 'Larry'],
'Marxes': ['Groucho', 'Chico', 'Harpo'],
'Pythons': ['Chapman', 'Cleese', 'Gilliam', 'Jones', 'Palin']}
```

在创建自定义数据结构的过程中，唯一的限制来自于这些内置数据类型本身。比如，字典的键必须为不可变对象，因此列表、字典以及集合都不能作为字典的键，但元组可以作为字典的键。举个例子，我们可以通过 GPS 坐标（纬度，经度，海拔；B.6 节可以看到更多与地图有关的例子）定位感兴趣的位置：

```
>>> houses = {
        (44.79, -93.14, 285): 'My House',
        (38.89, -77.03, 13): 'The White House'
        }
```

3.8 练习

在这一章，你见到了一些更为复杂的数据结构：列表、元组、字典和集合。你可以使用这些数据结构以及第 2 章提到的数据类型（数字和字符串）来表示现实生活中各种各样的元素。

(1) 创建一个叫作 years_list 的列表，存储从你出生的那一年到五岁那一年的年份。例如，如果你是 1980 年出生的，那么你的列表应该是 years_list = [1980, 1981, 1982, 1983, 1984, 1985]。

如果你现在还没到五岁却在阅读本书，那我真的没有什么可教你的了。

(2) 在 years_list 中，哪一年是你三岁生日那年？别忘了，你出生的第一年算 0 岁。

(3) 在 years_list 中，哪一年你的年纪最大？

(4) 创建一个名为 things 的列表，包含以下三个元素："mozzarella"、"cinderella" 和 "salmonella"。

(5) 将 things 中代表人名的字符串变成首字母大写形式，并打印整个列表。看看列表中的元素改变了吗？

(6) 将 things 中代表奶酪的元素全部改成大写，并打印整个列表。

(7) 将代表疾病的元素从 things 中删除，收好你得到的诺贝尔奖，并打印列表。

(8) 创建一个名为 surprise 的列表，包含以下三个元素："Groucho"、"Chico" 和 "Harpo"。

(9) 将 surprise 列表的最后一个元素变成小写，翻转过来，再将首字母变成大写。

(10) 创建一个名为 e2f 的英法字典并打印出来。这里提供一些单词对：dog 是 chien，cat 是 chat，walrus 是 morse。

(11) 使用你的仅包含三个词的字典 e2f 查询并打印出 walrus 对应的的法语词。

(12) 利用 e2f 创建一个名为 f2e 的法英字典。注意要使用 items 方法。

(13) 使用 f2e，查询并打印法语词 chien 对应的英文词。

(14) 创建并打印由 e2f 的键组成的英语单词集合。

(15) 建立一个名为 life 的多级字典。将下面这些字符串作为顶级键：'animals'、'plants' 以及 'others'。令 'animals' 键指向另一个字典，这个字典包含键 'cats'、'octopi' 以及 'emus'。令 'cat' 键指向一个字符串列表，这个列表包括 'Henri'、'Grumpy' 和 'Lucy'。让其余的键都指向空字典。

(16) 打印 life 的顶级键。

(17) 打印 life['animals'] 的全部键。

(18) 打印 life['animals']['cats'] 的值。

第 4 章

Python外壳：代码结构

在前三章中，我们用到很多关于数据的例子，但是没有对它们进行复杂的操作。大多数的示例代码都很短，并且使用交互式解释器进行解释执行。本章将介绍如何组织代码和数据。

许多计算机编程语言使用字符（例如花括号 { 和 }）或者关键字（例如 begin 和 end）来划分代码段。在这些语言中，使用一致的代码缩进可以增加代码的可读性，并且有很多便利的工具整理缩进代码。

在吉多 · 范 · 罗苏姆开始考虑设计 Python 语言时，他决定通过代码缩进来区分代码块结构，避免输入太多的花括号和关键字。Python 使用空白来区分代码结构，这是初学者需要注意的不同寻常的第一点，而且有其他语言开发经验的人会觉得奇怪。但使用 Python 一段时间后会觉得很自然，而且会习惯于编写简洁的代码来进行大量的编程工作。

4.1 使用#注释

注释是程序中会被 Python 解释器忽略的一段文本。通过使用注释，可以解释和明确 Python 代码的功能，记录将来要修改的地方，甚至写下你想写的任何东西。在 Python 中使用 # 字符标记注释，从 # 开始到当前行结束的部分都是注释。你可以把注释作为单独的一行，如下所示：

```
>>> # 60 sec/min * 60 min/hr * 24 hr/day
>>> seconds_per_day = 86400
```

也可以把注释和代码放在同一行：

```
>>> seconds_per_day = 86400 # 60 sec/min * 60 min/hr * 24 hr/day
```

有很多含义和叫法：hash、sharp、pound 或者 octothorpe[1]。无论你如何称呼它[2]，在 Python 代码中，注释的作用只对它出现的当前行有效。

Python 没有多行注释的符号。你需要明确地在注释部分的每一行开始处加上一个 #。

```
>>> # 尽管Python不会喜欢,但是我可以在这里讲任何东西
... # 因为我被“保护”
... # 令人敬畏的#号
...
>>>
```

然而，如果它出现在文本串中，将回归普通字符 # 的角色：

```
>>> print("No comment:  quotes make the # harmless")
No comment: quotes make the # harmless.
```

4.2　使用\连接

程序在合理的长度下是易读的。一行程序的（非强制性）最大长度建议为 80 个字符。如果你在该长度下写不完你的代码，可以使用*连接符* \（反斜线）。把它放在一行的结束位置，Python 仍然将其解释为同一行。

例如，假设想把一些短字符串拼接为一个长字符串，可以按照下面的步骤：

```
>>> alphabet = ''
>>> alphabet += 'abcdefg'
>>> alphabet += 'hijklmnop'
>>> alphabet += 'qrstuv'
>>> alphabet += 'wxyz'
```

或者，使用连接符一步就可以完成：

```
>>> alphabet = 'abcdefg' + \
...      'hijklmnop' + \
...      'qrstuv' + \
...      'wxyz'
```

在 Python 表达式占很多行的情况下，行连接符也是必需的：

```
>>> 1 + 2 +
  File "<stdin>", line 1
    1 + 2 +
          ^
SyntaxError: invalid syntax
>>> 1 + 2 + \
... 3
6
>>>
```

注 1：样子像在你身边的凶恶的八脚怪物。

注 2：嘘，别叫它，小心它回来找你。

4.3 使用if、elif和else进行比较

到目前为止，我们几乎一直在讨论数据结构。现在，我们将迈出探讨代码结构的第一步，将数据编排成代码（上一章讲到集合时已经见到一些例子，希望那些不会造成理解困难）。下面第一个例子是一个 Python 小程序，判断一个布尔变量 `disaster` 的值，然后打印输出合适的取值：

```
>>> disaster = True
>>> if disaster:
...     print("Woe!")
... else:
...     print("Whee!")
...
Woe!
>>>
```

程序中，`if` 和 `else` 两行是 Python 用来声明判断条件（本例中是 `disaster` 的值）是否满足的语句。`print()` 是将字符打印到屏幕的 Python 内建函数。

> 如果你之前使用过其他编程语言，不需要在 `if` 判断语句中加上圆括号，例如 `if (disaster == True)` 是没必要的，但在判断的末尾要加上冒号（`:`）。如果你像我一样有时会忘记输入冒号，这会导致 Python 解释器报错。

每一个 `print()` 在判断语句之后要缩进。我一般缩进四个空格。尽管你可以使用你喜欢的任何方式缩进，但是在同一代码段内最好使用一致的缩进——从每一行的左边开始使用相同数量的缩进字符。推荐的代码缩进风格 PEP-8（http://legacy.python.org/dev/peps/pep-0008/）同样使用四个空格。避免使用 Tab 字符或者 Tab 与 Space 混合的缩进风格，这样会使缩进字符的数量变得混乱。

上面的示例代码做了以下事情，在随后的内容中我还会更详细地介绍。

- 将布尔值 `True` 赋值给变量 `disaster`。
- 使用 `if` 和 `else` 进行条件比较判断，根据 `disaster` 的值执行不同的代码。
- 调用 `print()` 函数，在屏幕打印文本。

同时你可以根据需要进行多层判断语句的嵌套：

```
>>> furry = True
>>> small = True
>>> if furry:
...     if small:
...         print("It's a cat.")
...     else:
...         print("It's a bear!")
... else:
...     if small:
...         print("It's a skink!")
...     else:
...         print("It's a human. Or a hairless bear.")
```

```
...
It's a cat.
```

在 Python 中，代码缩进决定了 if 和 else 是如何配对的。在第一个判断 furry 中，因为 furry 的值是 True，所以程序跳转到执行判断 if small。我们之前将 small 赋值为 True，所以 if small 的值被估计为 True。因此程序会执行它的下一行，输出 It's a cat。

如果要检查超过两个条件，可以使用 if、elif（即 else if）和 else：

```
>>> color = "puce"
>>> if color == "red":
...     print("It's a tomato")
... elif color == "green":
...     print("It's a green pepper")
... elif color == "bee purple":
...     print("I don't know what it is, but only bees can see it")
... else:
...     print("I've never heard of the color", color)
...
I've never heard of the color puce
```

上面的例子中，我们使用了 == 作为判断相等的操作符，Python 中的比较操作符见下表。

相等	==
不等于	!=
小于	<
不大于	<=
大于	>
不小于	>=
属于	in...

这些操作符都返回布尔值 True 或者 False。让我们看看这些是如何执行的，首先对变量 x 赋值：

```
>>> x = 7
```

现在，测试一些例子：

```
>>> x == 5
False
>>> x == 7
True
>>> 5 < x
True
>>> x < 10
True
```

注意，两个等号（==）是用来判断相等的，而一个等号（=）是把某个值赋给一个变量。

如果你想同时进行多重比较判断，可以使用布尔操作符 and、or 或者 not 连接来决定最终表达式的布尔取值。

布尔操作符的优先级没有比较表达式的代码段高，也就是说，表达式要先计算然后再比较。在这个例子中，x 赋值为 7，5 < x 返回 True，x < 10 也同样返回 True，最终的结果就是 True and True：

```
>>> 5 <  x and x < 10
True
```

2.2.2 节中提到，避免混淆的最简单的办法是加花括号：

```
>>> ( 5 < x ) and ( x < 10)
True
```

下面是其他的一些例子：

```
>>> 5 < x or x < 10
True
>>> 5 < x and x > 10
False
>>> 5 < x and not x > 10
True
```

如果对同一个变量做多个 and 比较操作，Python 允许下面的用法：

```
>>> 5 < x < 10
True
```

这个表达式和 5 < x and x < 10 是一样的，你也可以使用更长的比较：

```
>>> 5 < x < 10 < 999
True
```

什么是真值（True）

如果表达式的返回类型不是布尔会发生什么？什么情况下 Python 会认为是 True 和 False？

一个成假赋值不一定明确表示为 False，下面的情况也会被认为是 False。

布尔	False
null 类型	None
整型	0
浮点型	0.0
空字符串	''
空列表	[]
空元组	()
空字典	{}
空集合	set()

剩下的都会被认为是 True。Python 程序中使用定义“真值”（在本例中是“假值”）的方式来判断数据结构是否为空以及成假条件：

```
>>> some_list = []
>>> if some_list:
```

```
...     print("There's something in here")
... else:
...     print("Hey, it's empty!")
...
Hey, it's empty!
```

如果你在判断一个表达式而不是一个简单的变量，Python 会先计算表达式的值，然后返回布尔型结果。所以，如果你输入以下的式子：

```
if color == "red":
```

Python 会判断 `color == "red"`。在例子中，把颜色赋值为字符串 `"puce"`，所以 `color == "red"` 是成假的，Python 转而执行下面的判断：

```
elif color == "green":
```

4.4　使用while进行循环

使用 `if`、`elif` 和 `else` 条件判断的例子是自顶向下执行的，但是有时候我们需要重复一些操作——*循环*。Python 中最简单的循环机制是 `while`。打开交互式解释器，执行下面的从 1 打印到 5 的简单循环：

```
>>> count = 1
>>> while count <= 5:
...     print(count)
...     count += 1
...
1
2
3
4
5
>>>
```

首先将变量 `count` 的值赋为 `1`，`while` 循环比较 `count` 的值和 `5` 的大小关系，如果 `count` 小于等于 `5` 的话继续执行。在循环内部，打印 `count` 变量的值，然后使用语句 `count += 1` 对 `count` 进行自增操作，返回到循环的开始位置，继续比较 `count` 和 `5` 的大小关系。现在，`count` 变量的值为 `2`，因此 `while` 循环内部的代码会被再次执行，`count` 值变为 `3` 。

在 `count` 从 `5` 自增到 `6` 之前循环一直进行。然后下次判断时，`count <= 5` 的条件不满足，`while` 循环结束。Python 跳到循环下面的代码。

4.4.1　使用break跳出循环

如果你想让循环在某一条件下停止，但是不确定在哪次循环跳出，可以在*无限循环*中声明 `break` 语句。这次，我们通过 Python 的 `input()` 函数从键盘输入一行字符串，然后将字符串首字母转化成大写输出。当输入的一行仅含有字符 `q` 时，跳出循环 ：

```
>>> while True:
...     stuff = input("String to capitalize [type q to quit]: ")
```

```
...     if stuff == "q":
...         break
...     print(stuff.capitalize())
...
String to capitalize [type q to quit]: test
Test
String to capitalize [type q to quit]: hey, it works
Hey, it works
String to capitalize [type q to quit]: q
>>>
```

4.4.2　使用continue跳到循环开始

有时我们并不想结束整个循环，仅仅想跳到下一轮循环的开始。下面是一个编造的例子：读入一个整数，如果它是奇数则输出它的平方数；如果是偶数则跳过。同样使用 q 来结束循环，代码中加上了适当的注释：

```
>>> while True:
...     value = input("Integer, please [q to quit]: ")
...     if value == 'q':      # 停止循环
...         break
...     number = int(value)
...     if number % 2 == 0:   # 判断偶数
...        continue
...     print(number, "squared is", number*number)
...
Integer, please [q to quit]: 1
1 squared is 1
Integer, please [q to quit]: 2
Integer, please [q to quit]: 3
3 squared is 9
Integer, please [q to quit]: 4
Integer, please [q to quit]: 5
5 squared is 25
Integer, please [q to quit]: q
>>>
```

4.4.3　循环外使用else

如果 while 循环正常结束（没有使用 break 跳出），程序将进入到可选的 else 段。当你使用循环来遍历检查某一数据结构时，找到满足条件的解使用 break 跳出；循环结束，即没有找到可行解时，将执行 else 部分代码段：

```
>>> numbers = [1, 3, 5]
>>> position = 0
>>> while position < len(numbers):
...     number = numbers[position]
...     if number % 2 == 0:
...         print('Found even number', number)
...         break
...     position += 1
... else:  #没有执行break
```

```
...     print('No even number found')
...
No even number found
```

> else 在此处的用法不是很直观，可以认为是循环中没有调用 break 后执行的检查。

4.5　使用for迭代

Python 频繁地使用*迭代器*。它允许在数据结构长度未知和具体实现未知的情况下遍历整个数据结构，并且支持迭代快速读写中的数据，以及允许不能一次读入计算机内存的数据流的处理。

下面这一遍历序列的方法是可行的：

```
>>> rabbits = ['Flopsy', 'Mopsy', 'Cottontail', 'Peter']
>>> current = 0
>>> while current < len(rabbits):
...     print(rabbits[current])
...     current += 1
...
Flopsy
Mopsy
Cottontail
Peter
```

但是，有一种更优雅的、Python 风格的遍历方式：

```
>>> for rabbit in rabbits:
...     print(rabbit)
...
Flopsy
Mopsy
Cottontail
Peter
```

列表（例如 rabbits）、字符串、元组、字典、集合等都是 Python 中*可迭代*的对象。元组或者列表在一次迭代过程产生一项，而字符串迭代会产生一个字符，如下所示：

```
>>> word = 'cat'
>>> for letter in word:
...     print(letter)
...
c
a
t
```

对一个字典（或者字典的 keys() 函数）进行迭代将返回字典中的键。在下面的例子中，字典的键为图板游戏 Clue（《妙探寻凶》）中牌的类型：

```
>>> accusation = {'room': 'ballroom', 'weapon': 'lead pipe',
                  'person': 'Col. Mustard'}
>>> for card in accusation:  # 或者是for card in accusation.keys():
...     print(card)
...
room
weapon
person
```

如果想对字典的值进行迭代，可以使用字典的 values() 函数：

```
>>> for value in accusation.values():
...     print(value)
...
ballroom
lead pipe
Col. Mustard
```

为了以元组的形式返回键值对，可以使用字典的 items() 函数：

```
>>> for item in accusation.items():
...     print(item)
...
('room', 'ballroom')
('weapon', 'lead pipe')
('person', 'Col. Mustard')
```

记住，元组只能被初始化一次，它的值是不能改变的。对于调用函数 items() 返回的每一个元组，将第一个返回值（键）赋给 card，第二个返回值（值）赋给 contents：

```
>>> for card, contents in accusation.items():
...     print('Card', card, 'has the contents', contents)
...
Card weapon has the contents lead pipe
Card person has the contents Col. Mustard
Card room has the contents ballroom
```

4.5.1 使用break跳出循环

在 for 循环中跳出的用法和在 while 循环中是一样的。

4.5.2 使用continue跳到循环开始

在一个循环中使用 continue 会跳到下一次的迭代开始，这一点和 while 循环也是类似的。

4.5.3 循环外使用else

类似于 while，for 循环也可以使用可选的 else 代码段，用来判断 for 循环是否正常结束（没有调用 break 跳出），否则会执行 else 段。

当你想确认之前的 for 循环是否正常跑完，增加 else 判断是有用的。下面的例子中，for 循环打印输出奶酪的名称，并且如果任一奶酪在商店中找到则跳出循环：

```
>>> cheeses = []
>>> for cheese in cheeses:
...     print('This shop has some lovely', cheese)
...     break
... else:  # 没有break表示没有找到奶酪
...     print('This is not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

在 for 循环外使用 else 可能和 while 循环一样，不是很直观和容易理解。下面的理解方式会更清楚：for 循环用来遍历查找，如果没有找到则调用执行 else。同样在没有 else 的情况下，为了达到相同的作用，可以声明某个变量指出在 for 循环中是否找到，看下面的例子：

```
>>> cheeses = []
>>> found_one = False
>>> for cheese in cheeses:
...     found_one = True
...     print('This shop has some lovely', cheese)
...     break
...
>>> if not found_one:
...     print('This is not much of a cheese shop, is it?')
...
This is not much of a cheese shop, is it?
```

4.5.4　使用zip()并行迭代

在使用迭代时，有一个非常方便的技巧：通过 zip() 函数对多个序列进行并行迭代：

```
>>> days = ['Monday', 'Tuesday', 'Wednesday']
>>> fruits = ['banana', 'orange', 'peach']
>>> drinks = ['coffee', 'tea', 'beer']
>>> desserts = ['tiramisu', 'ice cream', 'pie', 'pudding']
>>> for day, fruit, drink, dessert in zip(days, fruits, drinks, desserts):
...     print(day, ": drink", drink, "- eat", fruit, "- enjoy", dessert)
...
Monday : drink coffee - eat banana - enjoy tiramisu
Tuesday : drink tea - eat orange - enjoy ice cream
Wednesday : drink beer - eat peach - enjoy pie
```

zip() 函数在最短序列“用完”时就会停止。上面例子中的列表（desserts）是最长的，所以我们无法填充列表，除非人工扩展其他列表。

3.4 节中的 dict() 函数会将两项序列，比如元组、列表、字符串，创建成一个字典，同时使用 zip() 函数可以遍历多个序列，在具有相同位移的项之间创建元组。下面创建英语单词和法语单词之间的对应关系的两个元组：

```
>>> english = 'Monday', 'Tuesday', 'Wednesday'
>>> french = 'Lundi', 'Mardi', 'Mercredi'
```

现在使用 zip() 函数配对两个元组。函数的返回值既不是元组也不是列表，而是一个整合

在一起的可迭代变量：

```
>>> list( zip(english, french) )
[('Monday', 'Lundi'), ('Tuesday', 'Mardi'), ('Wednesday', 'Mercredi')]
```

配合 dict() 函数和 zip() 函数的返回值就可以得到一本微型的英法词典：

```
>>> dict( zip(english, french) )
{'Monday': 'Lundi', 'Tuesday': 'Mardi', 'Wednesday': 'Mercredi'}
```

4.5.5　使用range()生成自然数序列

range() 函数返回在特定区间的自然数序列，不需要创建和存储复杂的数据结构，例如列表或者元组。这允许在不使用计算机全部内存的情况下创建较大的区间，也不会使你的程序崩溃。

range() 函数的用法类似于使用切片：range（start、stop、step）。而 start 的默认值为 0。唯一要求的参数值是 stop，产生的最后一个数的值是 stop 的前一个，并且 step 的默认值是 1。当然，也可以反向创建自然数序列，这时 step 的值为 -1。

像 zip()、range() 这些函数返回的是一个可迭代的对象，所以可以使用 for ... in 的结构遍历，或者把这个对象转化为一个序列（例如列表）。我们来产生序列 0，1，2：

```
>>> for x in range(0,3):
...     print(x)
...
0
1
2
>>> list( range(0, 3) )
[0, 1, 2]
```

下面是如何从 2 到 0 反向创建序列：

```
>>> for x in range(2, -1, -1):
...     print(x)
...
2
1
0
>>> list( range(2, -1, -1) )
[2, 1, 0]
```

下面的代码片段将 step 赋值为 2，得到从 0 到 10 的偶数：

```
>>> list( range(0, 11, 2) )
[0, 2, 4, 6, 8, 10]
```

4.5.6　其他迭代方式

第 8 章将介绍文件之间的迭代。在第 6 章中，你会看到如何在自己创建的对象之间迭代。

4.6 推导式

*推导式*是从一个或者多个迭代器快速简洁地创建数据结构的一种方法。它可以将循环和条件判断结合，从而避免语法冗长的代码。会使用推导式有时可以说明你已经超过 Python 初学者的水平。也就是说，使用推导式更像 Python 风格。

4.6.1 列表推导式

你可以从 `1` 到 `5` 创建一个整数列表，每次增加一项，如下所示：

```
>>> number_list = []
>>> number_list.append(1)
>>> number_list.append(2)
>>> number_list.append(3)
>>> number_list.append(4)
>>> number_list.append(5)
>>> number_list
[1, 2, 3, 4, 5]
```

或者，可以结合 `range()` 函数使用一个迭代器：

```
>>> number_list = []
>>> for number in range(1, 6):
...     number_list.append(number)
...
>>> number_list
[1, 2, 3, 4, 5]
```

或者，直接把 `range()` 的返回结果放到一个列表中：

```
>>> number_list = list(range(1, 6))
>>> number_list
[1, 2, 3, 4, 5]
```

上面这些方法都是可行的 Python 代码，会得到相同的结果。然而，更像 Python 风格的创建列表方式是使用*列表推导*。最简单的形式如下所示：

`[` *`expression`* `for` *`item`* `in` *`iterable`* `]`

下面的例子将通过列表推导创建一个整数列表：

```
>>> number_list = [number for number in range(1,6)]
>>> number_list
[1, 2, 3, 4, 5]
```

在第一行中，第一个 `number` 变量为列表生成值，也就是说，把循环的结果放在列表 `number_list` 中。第二个 `number` 为循环变量。其中第一个 `number` 可以为表达式，试试下面改编的例子：

```
>>> number_list = [number-1 for number in range(1,6)]
>>> number_list
[0, 1, 2, 3, 4]
```

列表推导把循环放在方括号内部。这种例子和之前碰到的不大一样，但却是更为常见的方式。同样，列表推导也可以像下面的例子加上条件表达式：

`[`*`expression`*` for `*`item`*` in `*`iterable`*` if `*`condition`*`]`

现在，通过推导创建一个在 1 到 5 之间的偶数列表（当 `number % 2` 为真时，代表奇数；为假时，代表偶数）：

```
>>> a_list = [number for number in range(1,6) if number % 2 == 1]
>>> a_list
[1, 3, 5]
```

于是，上面的推导要比之前传统的方法更简洁：

```
>>> a_list = []
>>> for number in range(1,6):
...     if number % 2 == 1:
...         a_list.append(number)
...
>>>  a_list
[1, 3, 5]
```

最后，正如存在很多嵌套循环一样，在对应的推导中会有多个 `for ...` 语句。我们先来看一个简单的嵌套循环的例子，并在屏幕上打印出结果：

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> for row in rows:
...     for col in cols:
...         print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

下面使用一次推导，将结果赋值给变量 `cells`，使它成为元组 `(row,col)`：

```
>>> rows = range(1,4)
>>> cols = range(1,3)
>>> cells = [(row, col) for row in rows for col in cols]
>>> for cell in cells:
...     print(cell)
...
(1, 1)
(1, 2)
(2, 1)
(2, 2)
(3, 1)
(3, 2)
```

另外，在对 `cells` 列表进行迭代时可以通过元组拆封将变量 `row` 和 `col` 的值分别取出：

```
>>> for row, col in cells:
...     print(row, col)
...
1 1
1 2
2 1
2 2
3 1
3 2
```

其中，在列表推导中 `for row ...` 和 `for col ...` 都可以有自己单独的 `if` 条件判断。

4.6.2 字典推导式

除了列表，字典也有自己的推导式。最简单的例子就像：

`{ key_expression : value_expression for expression in iterable }`

类似于列表推导，字典推导也有 `if` 条件判断以及多个 `for` 循环迭代语句：

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in word}
>>> letter_counts
{'l': 1, 'e': 2, 't': 2, 'r': 1, 's': 1}
```

程序中，对字符串 `'letters'` 中出现的字母进行循环，计算出每个字母出现的次数。对于程序执行来说，两次调用 `word.count(letter)` 浪费时间，因为字符串中 `t` 和 `e` 都出现了两次，第一次调用 `word.count()` 时已经计算得到相应的值。下面的例子会解决这个小问题，更符合 Python 风格：

```
>>> word = 'letters'
>>> letter_counts = {letter: word.count(letter) for letter in set(word)}
>>> letter_counts
{'t': 2, 'l': 1, 'e': 2, 'r': 1, 's': 1}
```

字典键的顺序和之前的例子是不同的，因为是对 `set(word)` 集合进行迭代的，而前面的例子是对 `word` 字符串迭代。

4.6.3 集合推导式

集合也不例外，同样有推导式。最简单的版本和之前的列表、字典推导类似：

`{expression for expression in iterable }`

最长的版本（`if` tests, multiple `for` clauses）对于集合而言也是可行的：

```
>>> a_set = {number for number in range(1,6) if number % 3 == 1}
>>> a_set
{1, 4}
```

4.6.4 生成器推导式

元组是没有推导式的。你可能认为将列表推导式中的方括号变成圆括号就可以定义元组推

导式，就像下面的表达式一样：

```
>>> number_thing = (number for number in range(1, 6))
```

其实，圆括号之间的是*生成器推导式*，它返回的是一个*生成器对象*：

```
>>> type(number_thing)
<class 'generotor'>
```

4.8 节会详细介绍。它是将数据传给迭代器的一种方式。

你可以直接对生成器对象进行迭代，如下所示：

```
>>> for number in number_thing:
...     print(number)
...
1
2
3
4
5
```

或者，通过对一个生成器的推导式调用 `list()` 函数，使它类似于列表推导式：

```
>>> number_list = list(number_thing)
>>> number_list
[1, 2, 3, 4, 5]
```

一个生成器只能运行一次。列表、集合、字符串和字典都存储在内存中，但是生成器仅在运行中产生值，不会被存下来，所以不能重新使用或者备份一个生成器。

如果想再一次迭代此生成器，会发现它被擦除了：

```
>>> try_again = list(number_thing)
>>> try_again
[]
```

你既可以通过生成器推导式创建生成器，也可以使用*生成器的函数*。后面会介绍这些函数，并且探讨这些生成器函数的特殊用法。

4.7　函数

到目前为止，我们的 Python 代码已经实现了小的分块。它们都适合处理微小任务，但是我们想复用这些代码，所以需要把大型代码组织成可管理的代码段。

代码复用的第一步是使用*函数*，它是命名的用于区分的代码段。函数可以接受任何数字或者其他类型的输入作为*参数*，并且返回数字或者其他类型的结果。

你可以使用函数做以下两件事情：

- 定义函数

• 调用函数

为了定义 Python 函数，你可以依次输入 def、函数名、带有函数参数的圆括号，最后紧跟一个冒号（:）。函数命名规范和变量命名一样（必须使用字母或者下划线 _ 开头，仅能含有字母、数字和下划线）。

我们先定义和调用一个没有参数的函数。下面的例子是最简单的 Python 函数：

```
>>> def do_nothing():
...     pass
```

即使对于一个没有参数的函数，仍然需要在定义时加上圆括号和冒号。下面的一行需要像声明 if 语句一样缩进。Python 函数中的 pass 表明函数没有做任何事情。和这一页故意留白有同样的作用（即使它不再是）。

通过输入函数名和参数调用此函数，像前面说的一样，它没有做任何事情：

```
>>> do_nothing()
>>>
```

现在，定义一个无参数，但打印输出一个单词的函数：

```
>>> def make_a_sound():
...     print('quack')
...
>>> make_a_sound()
quack
```

当调用 make_a_sound() 函数时，Python 会执行函数内部的代码。在这个例子中，函数打印输出单个词，并且返回到主程序。

下面尝试一个没有参数但返回值的函数：

```
>>> def agree():
...    return True
...
```

或者，调用这个函数，使用 if 语句检查它的返回值：

```
>>> if agree():
...     print('Splendid!')
... else:
...     print('That was unexpected.')
...
Splendid!
```

学到现在已经迈出了很大的一步。在函数中，使用 if 判断和 for/while 循环组合能实现之前无法实现的功能。

这个时候该在函数中引入参数。定义带有一个 anything 参数的函数 echo()。它使用 return 语句将 anything 返回给它的调用者两次，并在两次中间加入一个空格：

```
>>> def echo(anything):
...    return anything + ' ' + anything
```

```
...
>>>
```

然后用字符串 'Rumplestiltskin' 调用函数 echo()：

```
>>> echo('Rumplestiltskin')
'Rumplestiltskin Rumplestiltskin'
```

传入到函数的值称为参数。当调用含参数的函数时，这些参数的值会被复制给函数中的对应参数。在之前的例子中，被调用的函数 echo() 的传入参数字符串是 'Rumplestiltskin'，这个值被复制给参数 anything，然后返回到调用方（在这个例子中，输出两次字符串，中间有一个空格）。

上面的这些函数例子都很基础。现在我们写一个含有输入参数的函数，它能真正处理一些事情。在这里依旧沿用评论颜色的代码段。调用 commentary 函数，把 color 作为输入的参数，使它返回对颜色的评论字符串：

```
>>> def commentary(color):
...     if color == 'red':
...         return "It's a tomato."
...     elif color == "green":
...         return "It's a green pepper."
...     elif color == 'bee purple':
...         return "I don't know what it is, but only bees can see it."
...     else:
...         return "I've never heard of the color "  + color +  "."
...
>>>
```

传入字符串参数 'blue'，调用函数 commentary()：

```
>>> comment = commentary('blue')
```

这个函数做了以下事情：

- 把 'blue' 赋值给函数的内部参数 color
- 运行 if-elif-else 的逻辑链
- 返回一个字符串
- 将该字符串赋值给变量 comment

我们如何得到返回值呢？

```
>>> print(comment)
I've never heard of the color blue.
```

一个函数可以接受任何数量（包括 0）的任何类型的值作为输入变量，并且返回任何数量（包括 0）的任何类型的结果。如果函数不显式调用 return 函数，那么会默认返回 None。

```
>>> print(do_nothing())
None
```

有用的 None

None 是 Python 中一个特殊的值，虽然它不表示任何数据，但仍然具有重要的作用。虽然 None 作为布尔值和 False 是一样的，但是它和 False 有很多差别。下面是一个例子：

```
>>> thing = None
>>> if thing:
...     print("It's some thing")
... else:
...     print("It's no thing")
...
It's no thing
```

为了区分 None 和布尔值 False，使用 Python 的 is 操作符：

```
>>> if thing is None:
...     print("It's nothing")
... else:
...     print("It's something")
...
It's nothing
```

这虽然是一个微妙的区别，但是对于 Python 来说是很重要的。你需要把 None 和不含任何值的空数据结构区分开来。0 值的整型 / 浮点型、空字符串（''）、空列表（[]）、空元组（(,)）、空字典（{}）、空集合（set()）都等价于 False，但是不等于 None。

现在，快速写一个函数，输出它的参数是否是 None：

```
>>> def is_none(thing):
...     if thing is None:
...         print("It's None")
...     elif thing:
...         print("It's True")
...     else:
...         print("It's False")
...
```

现在，运行一些测试函数：

```
>>> is_none(None)
It's None
>>> is_none(True)
It's True
>>> is_none(False)
It's False
>>> is_none(0)
It's False
>>> is_none(0.0)
It's False
>>> is_none(())
It's False
```

```
>>> is_none([])
It's False
>>> is_none({})
It's False
>>> is_none(set())
It's False
```

4.7.1 位置参数

Python 处理参数的方式要比其他语言更加灵活。其中，最熟悉的参数类型是*位置参数*，传入参数的值是按照顺序依次复制过去的。

下面创建一个带有位置参数的函数，并且返回一个字典：

```
>>> def menu(wine, entree, dessert):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
...
>>> menu('chardonnay', 'chicken', 'cake')
{'dessert': 'cake', 'wine': 'chardonnay', 'entree': 'chicken'}
```

尽管这种方式很常见，但是位置参数的一个弊端是必须熟记每个位置的参数的含义。在调用函数 `menu()` 时误把最后一个参数当作第一个参数，会得到完全不同的结果：

```
>>> menu('beef', 'bagel', 'bordeaux')
{'dessert': 'bordeaux', 'wine': 'beef', 'entree': 'bagel'}
```

4.7.2 关键字参数

为了避免位置参数带来的混乱，调用参数时可以指定对应参数的名字，甚至可以采用与函数定义不同的顺序调用：

```
>>> menu(entree='beef', dessert='bagel', wine='bordeaux')
{'dessert': 'bagel', 'wine': 'bordeaux', 'entree': 'beef'}
```

你也可以把位置参数和关键字参数混合起来。首先，实例化参数 `wine`，然后对参数 `entree` 和 `dessert` 使用关键字参数的方式：

```
>>> menu('frontenac', dessert='flan', entree='fish')
{'entree': 'fish', 'dessert': 'flan', 'wine': 'frontenac'}
```

如果同时出现两种参数形式，首先应该考虑的是位置参数。

4.7.3 指定默认参数值

当调用方没有提供对应的参数值时，你可以指定默认参数值。这个听起来很普通的特性实际上特别有用，以之前的例子为例：

```
>>> def menu(wine, entree, dessert='pudding'):
...     return {'wine': wine, 'entree': entree, 'dessert': dessert}
```

这一次调用不带 `dessert` 参数的函数 `menu()`：

```
>>> menu('chardonnay', 'chicken')
{'dessert': 'pudding', 'wine': 'chardonnay', 'entree': 'chicken'}
```

如果你提供参数值，在调用时会代替默认值：

```
>>> menu('dunkelfelder', 'duck', 'doughnut')
{'dessert': 'doughnut', 'wine': 'dunkelfelder', 'entree': 'duck'}
```

默认参数值在函数被定义时已经计算出来，而不是在程序运行时。Python 程序员经常犯的一个错误是把可变的数据类型（例如列表或者字典）当作默认参数值。

在下面的例子中，函数 buggy() 在每次调用时，添加参数 arg 到一个空的列表 result，然后打印输出一个单值列表。但是存在一个问题：只有在第一次调用时列表是空的，第二次调用时就会存在之前调用的返回值：

```
>>> def buggy(arg, result=[]):
...     result.append(arg)
...     print(result)
...
>>> buggy('a')
['a']
>>> buggy('b')   # expect ['b']
['a', 'b']
```

如果写成下面的样子就会解决刚才的问题：

```
>>> def works(arg):
...     result = []
...     result.append(arg)
...     return result
...
>>> works('a')
['a']
>>> works('b')
['b']
```

这样的修改也为了表明是第一次调用跳过一些操作：

```
>>> def nonbuggy(arg, result=None):
...     if result is None:
...         result = []
...     result.append(arg)
...     print(result)
...
>>> nonbuggy('a')
['a']
>>> nonbuggy('b')
['b']
```

4.7.4　使用*收集位置参数

如果你之前使用 C/C++ 编程，可能会认为 Python 中的星号（*）和指针相关。然而，

Python 是没有指针概念的。

当参数被用在函数内部时，星号将一组可变数量的位置参数集合成参数值的元组。在下面的例子中 args 是传入到函数 print_args() 的参数值的元组：

```
>>> def print_args(*args):
...     print('Positional argument tuple:', args)
...
```

无参数调用函数，则什么也不会返回：

```
>>> print_args()
Positional argument tuple: ()
```

给函数传入的所有参数都会以元组的形式返回输出：

```
>>> print_args(3, 2, 1, 'wait!', 'uh...')
Positional argument tuple: (3, 2, 1, 'wait!', 'uh...')
```

这样的技巧对于编写像 print() 一样接受可变数量的参数的函数是非常有用的。如果你的函数同时有限定的位置参数，那么 *args 会收集剩下的参数：

```
>>> def print_more(required1, required2, *args):
...     print('Need this one:', required1)
...     print('Need this one too:', required2)
...     print('All the rest:', args)
...
>>> print_more('cap', 'gloves', 'scarf', 'monocle', 'mustache wax')
Need this one: cap
Need this one too: gloves
All the rest: ('scarf', 'monocle', 'mustache wax')
```

当使用 * 时不需要调用元组参数 args，不过这也是 Python 的一个常见做法。

4.7.5 使用**收集关键字参数

使用两个星号可以将参数收集到一个字典中，参数的名字是字典的键，对应参数的值是字典的值。下面的例子定义了函数 print_kwargs()，然后打印输出它的关键字参数：

```
>>> def print_kwargs(**kwargs):
...     print('Keyword arguments:', kwargs)
...
```

现在，使用一些关键字参数调用函数：

```
>>> print_kwargs(wine='merlot', entree='mutton', dessert='macaroon')
Keyword arguments: {'dessert': 'macaroon', 'wine': 'merlot', 'entree': 'mutton'}
```

在函数内部，kwargs 是一个字典。

如果你把带有 *args 和 **kwargs 的位置参数混合起来，它们会按照顺序解析。和 args 一样，调用时不需要参数 kwargs，这也是常见用法。

4.7.6 文档字符串

正如《Python 之禅》（*the Zen of Python*）中提到的，程序的可读性很重要。建议在函数体开始的部分附上函数定义说明的文档，这就是函数的文档字符串：

```
>>> def echo(anything):
...     'echo returns its input argument'
...     return anything
```

可以定义非常长的文档字符串，加上详细的规范说明，如下所示：

```
def print_if_true(thing, check):
    '''
    Prints the first argument if a second argument is true.
    The operation is:
        1. Check whether the *second* argument is true.
        2. If it is, print the *first* argument.
    '''
    if check:
        print(thing)
```

调用 Python 函数 `help()` 可以打印输出一个函数的文档字符串。把函数名传入函数 `help()` 就会得到参数列表和规范的文档：

```
>>> help(echo)
Help on function echo in module __main__:

echo(anything)
    echo returns its input argument
```

如果仅仅想得到文档字符串：

```
>>> print(echo.__doc__)
echo returns its input argument
```

看上去很奇怪的 `__doc__` 是作为函数中变量的文档字符串的名字。4.10 节的“名称中 _ 和 __ 的用法”会解释所有下划线背后的原理。

4.7.7 一等公民：函数

之前提过 Python 中一切都是对象，包括数字、字符串、元组、列表、字典和函数。函数是 Python 中的一等公民，可以把它们（返回值）赋给变量，可以作为参数被其他函数调用，也可以从其他函数中返回值。它可以帮助你在 Python 中实现其他语言难以实现的功能。

为了测试，现在定义一个简单的函数 `answer()`，它没有任何参数，仅仅打印输出数字 `42`：

```
>>> def answer():
...     print(42)
```

运行该函数，会得到下面的结果：

```
>>> answer()
42
```

再定义一个函数 run_something。它有一个参数 func，这个参数是一个可以运行的函数的名字：

```
>>> def run_something(func):
...     func()
```

将参数 answer 传到该函数，在这里像之前碰到的一样，把函数名当作数据使用：

```
>>> run_something(answer)
42
```

注意，你传给函数的是 answer，而不是 answer()。在 Python 中圆括号意味着调用函数。在没有圆括号的情况下，Python 会把函数当作普通对象。这是因为在其他情况下，它也仅仅代表一个对象：

```
>>> type(run_something)
<class 'function'>
```

我们来运行一个带参数的例子。定义函数 add_args()，它会打印输出两个数值参数（arg1 和 arg2）的和：

```
>>> def add_args(arg1, arg2):
...     print(arg1 + arg2)
```

那么，add_args() 的类型是什么？

```
>>> type(add_args)
<class 'function'>
```

此刻定义一个函数 run_something_with_args()，它带有三个参数：

- func——可以运行的函数
- arg1——func 函数的第一个参数
- arg2——func 函数的第二个参数

```
>>> def run_something_with_args(func, arg1, arg2):
...     func(arg1, arg2)
```

当调用 run_something_with_args() 时，调用方传来的函数赋值给 func 参数，而 arg1 和 arg2 从参数列表中获得值。然后运行带参数的 func(arg1, arg2)。

将函数名 add_args 和参数 5、9 传给函数 run_something_with_args()：

```
>>> run_something_with_args(add_args, 5, 9)
14
```

在函数 run_something_with_args() 内部，函数名 add_args 被赋值给参数 func，5 和 9 分别赋值给 arg1 和 arg2。程序最后执行：

```
add_args(5, 9)
```

同样可以在此用上 *args（位置参数收集）和 **kwargs（关键字参数收集）的技巧。

我们定义一个测试函数，它可以接受任意数量的位置参数，使用 sum() 函数计算它们的和，

并返回这个和：

```
>>> def sum_args(*args):
...    return sum(args)
```

之前没有提到 sum() 函数，它是 Python 的一个内建函数，用来计算可迭代的数值（整型或者浮点型）参数的和。

下面再定义一个新的函数 run_with_positional_args()，接收一个函数名以及任意数量的位置参数：

```
>>> def run_with_positional_args(func, *args):
...    return func(*args)
```

现在直接调用它：

```
>>> run_with_positional_args(sum_args, 1, 2, 3, 4)
10
```

同样可以把函数作为列表、元组、集合和字典的元素。函数名是不可变的，因此可以把函数用作字典的键。

4.7.8　内部函数

在 Python 中，可以在函数中定义另外一个函数：

```
>>> def outer(a, b):
...     def inner(c, d):
...         return c + d
...     return inner(a, b)
...
>>>
>>> outer(4, 7)
11
```

当需要在函数内部多次执行复杂的任务时，内部函数是非常有用的，从而避免了循环和代码的堆叠重复。对于这样一个字符串的例子，内部函数的作用是给外部的函数增加字符串参数：

```
>>> def knights(saying):
...     def inner(quote):
...         return "We are the knights who say: '%s'" % quote
...     return inner(saying)
...
>>> knights('Ni!')
"We are the knights who say: 'Ni!'"
```

4.7.9　闭包

内部函数可以看作一个闭包。闭包是一个可以由另一个函数动态生成的函数，并且可以改变和存储函数外创建的变量的值。

下面的例子以之前的 knights() 为基础。现在，调用新的函数 knight2()，把 inner() 函数变成一个叫 inner2() 的闭包。可以看出有以下不同点。

- inner2() 直接使用外部的 saying 参数，而不是通过另外一个参数获取。
- knights2() 返回值为 inner2 函数，而不是调用它。

```
>>> def knights2(saying):
...     def inner2():
...         return "We are the knights who say: '%s'" % saying
...     return inner2
...
```

inner2() 函数可以得到 saying 参数的值并且记录下来。return inner2 这一行返回的是 inner2 函数的复制（没有直接调用）。所以它就是一个闭包：一个被动态创建的可以记录外部变量的函数。

用不同的参数调用 knights2() 两次：

```
>>> a = knights2('Duck')
>>> b = knights2('Hasenpfeffer')
```

那么 a 和 b 会是什么类型？

```
>>> type(a)
<class 'function'>
>>> type(b)
<class 'function'>
```

它们是函数，同时也是闭包：

```
>>> a
<function knights2.<locals>.inner2 at 0x10193e158>
>>> b
<function knights2.<locals>.inner2 at 0x10193e1e0>
```

如果调用它们，它们会记录被 knights2 函数创建时的外部变量 saying：

```
>>> a()
"We are the knights who say: 'Duck'"
>>> b()
"We are the knights who say: 'Hasenpfeffer'"
```

4.7.10 匿名函数：lambda()函数

Python 中，lambda 函数是用一个语句表达的匿名函数。可以用它来代替小的函数。

首先，举一个使用普通函数的例子。定义函数 edit_story()，参数列表如下所示：

- words——单词列表
- func——遍历列表中单词的函数

```
>>> def edit_story(words, func):
...     for word in words:
...         print(func(word))
```

现在，需要一个单词列表和一个遍历单词的函数。对于单词，可以选择我的猫从某一台阶上掉下时发出的声音：

```
>>> stairs = ['thud', 'meow', 'thud', 'hiss']
```

对于函数，它要将每个单词的首字母变为大写，然后在末尾加上感叹号，用作猫画报的标题非常完美：

```
>>> def enliven(word):   # 让这些单词更有情感
...     return word.capitalize() + '!'
```

混合这些“配料”：

```
>>> edit_story(stairs, enliven)
Thud!
Meow!
Thud!
Hiss!
```

最后，到了 lambda。`enliven()` 函数可以简洁地用下面的一个 lambda 代替：

```
>>>
>>> edit_story(stairs, lambda word: word.capitalize() + '!')
Thud!
Meow!
Thud!
Hiss!
>>>
```

lambda 函数接收一个参数 `word`。在冒号和末尾圆括号之间的部分为函数的定义。

通常，使用实际的函数（例如 `enliven()`）会比使用 lambda 更清晰明了。但是，当需要定义很多小的函数以及记住它们的名字时，lambda 会非常有用。尤其是在图形用户界面中，可以使用 lambda 来定义*回调函数*。请参见附录 A 中的例子。

4.8　生成器

*生成器*是用来创建 Python 序列的一个对象。使用它可以迭代庞大的序列，且不需要在内存中创建和存储整个序列。通常，生成器是为迭代器产生数据的。回想起来，我们已经在之前的例子中使用过其中一个，即 `range()`，来产生一系列整数。`range()` 在 Python 2 中返回一个列表，这也限制了它要进入内存空间。Python 2 中同样存在的生成器 `xrange()` 在 Python 3 中成为标准的 `range()` 生成器。这个例子累加从 1 到 100 的整数：

```
>>> sum(range(1, 101))
5050
```

每次迭代生成器时，它会记录上一次调用的位置，并且返回下一个值。这一点和普通的函数是不一样的，一般函数都不记录前一次调用，而且都会在函数的第一行开始执行。

如果你想创建一个比较大的序列，使用生成器推导的代码会很长，这时可以尝试写一个*生成器函数*。生成器函数和普通函数类似，但是它的返回值使用 `yield` 语句声明而不是

return。下面编写我们自己的 range() 函数版本：

```
>>> def my_range(first=0, last=10, step=1):
...     number = first
...     while number < last:
...         yield number
...         number += step
...
```

这是一个普通的函数：

```
>>> my_range
<function my_range at 0x10193e268>
```

并且它返回的是一个生成器对象：

```
>>> ranger = my_range(1, 5)
>>> ranger
<generator object my_range at 0x101a0a168>
```

可以对这个生成器对象进行迭代：

```
>>> for x in ranger:
...        print(x)
...
1
2
2
4
```

4.9　装饰器

有时你需要在不改变源代码的情况下修改已经存在的函数。常见的例子是增加一句调试声明，以查看传入的参数。

装饰器实质上是一个函数。它把一个函数作为输入并且返回另外一个函数。在装饰器中，通常使用下面这些 Python 技巧：

- *args 和 **kwargs
- 闭包
- 作为参数的函数

函数 document_it() 定义了一个装饰器，会实现如下功能：

- 打印输出函数的名字和参数的值
- 执行含有参数的函数
- 打印输出结果
- 返回修改后的函数

看下面的代码：

```
>>> def document_it(func):
...     def new_function(*args, **kwargs):
```

```
...         print('Running function:', func.__name__)
...         print('Positional arguments:', args)
...         print('Keyword arguments:', kwargs)
...         result = func(*args, **kwargs)
...         print('Result:', result)
...         return result
...     return new_function
```

无论传入 document_it() 的函数 func 是什么，装饰器都会返回一个新的函数，其中包含函数 document_it() 增加的额外语句。实际上，装饰器并不需要执行函数 func 中的代码，只是在结束前函数 document_it() 调用函数 func 以便得到 func 的返回结果和附加代码的结果。

那么，如何使用装饰器？当然，可以通过人工赋值：

```
>>> def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
8
>>> cooler_add_ints = document_it(add_ints)  # 人工对装饰器赋值
>>> cooler_add_ints(3, 5)
Running function: add_ints
Postitional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

作为对前面人工装饰器赋值的替代，可以直接在要装饰的函数前添加装饰器名字 @decorator_name：

```
>>> @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Start function add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
8
```

同样一个函数可以有多个装饰器。下面，我们写一个对结果求平方的装饰器 square_it()：

```
>>> def square_it(func):
...     def new_function(*args, **kwargs):
...         result = func(*args, **kwargs)
...         return result * result
...     return new_function
...
```

靠近函数定义（def 上面）的装饰器最先执行，然后依次执行上面的。任何顺序都会得到相同的最终结果。下面的例子中会看到中间步骤的变化：

```
>>> @document_it
... @square_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: new_function
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 64
64
```

交换两个装饰器的顺序：

```
>>> @square_it
... @document_it
... def add_ints(a, b):
...     return a + b
...
>>> add_ints(3, 5)
Running function: add_ints
Positional arguments: (3, 5)
Keyword arguments: {}
Result: 8
64
```

4.10　命名空间和作用域

一个名称在不同的使用情况下可能指代不同的事物。Python 程序有各种各样的命名空间，它指的是在该程序段内一个特定的名称是独一无二的，它和其他同名的命名空间是无关的。

每一个函数定义自己的命名空间。如果在主程序（main）中定义一个变量 x，在另外一个函数中也定义 x 变量，两者指代的是不同的变量。但是，天下也没有完全绝对的事情，需要的话，可以通过多种方式获取其他命名空间的名称。

每个程序的主要部分定义了*全局*命名空间。因此，在这个命名空间的变量是*全局变量*。

你可以在一个函数内得到某个全局变量的值：

```
>>> animal = 'fruitbat'
>>> def print_global():
...     print('inside print_global:', animal)
...
>>> print('at the top level:', animal)
at the top level: fruitbat
>>> print_global()
inside print_global: fruitbat
```

但是，如果想在函数中得到一个全局变量的值并且改变它，会报错：

```
>>> def change_and_print_global():
...     print('inside change_and_print_global:', animal)
...     animal = 'wombat'
```

```
...     print('after the change:', animal)
...
>>> change_and_print_global()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in change_and_report_it
UnboundLocalError: local variable 'animal' referenced before assignment
```

实际上，你改变的另外一个同样被命名为 animal 的变量，只不过这个变量在函数内部：

```
>>> def change_local():
...     animal = 'wombat'
...     print('inside change_local:', animal, id(animal))
...
>>> change_local()
inside change_local: wombat 4330406160
>>> animal
'fruitbat'
>>> id(animal)
4330390832
```

这里发生了什么？在函数第一行将字符串 fruitbat 赋值给全局变量 animal。函数 change_local() 也有一个叫作 animal 的变量。不同的是，它在自己的局部命名空间。

我们使用 Python 内嵌函数 id() 打印输出每个对象的唯一的 ID 值，证明在函数 change_local() 中的变量 animal 和主程序中的 animal 不是同一个。

为了读取全局变量而不是函数中的局部变量，需要在变量前面显式地加关键字 global（也正是《Python 之禅》中的一句话：明了胜于隐晦）：

```
>>> animal = 'fruitbat'
>>> def change_and_print_global():
...     global animal
...     animal = 'wombat'
...     print('inside change_and_print_global:', animal)
...
>>> animal
'fruitbat'
>>> change_and_print_global()
inside change_and_print_global: wombat
>>> animal
'wombat'
```

如果在函数中不声明关键字 global，Python 会使用局部命名空间，同时变量也是局部的。函数执行后回到原来的命名空间。

Python 提供了两个获取命名空间内容的函数：

- locals() 返回一个局部命名空间内容的字典；
- globals() 返回一个全局命名空间内容的字典。

下面是它们的实例：

```
>>> animal = 'fruitbat'
>>> def change_local():
```

```
...     animal = 'wombat'   #局部变量
...     print('locals:',locals())
...
>>> animal
'fruitbat'
>>> change_local()
locals: {'animal':'wombat'}
>>> print('globals:', globals())  #表示时格式稍微发生变化
globals:{'animal': 'fruitbat',
'__doc__': None,
'change_local': <function change_it at 0x1006c0170>,
'__package__': None,
'__name__': '__main__',
'__loader__': <class '_frozen_importlib.BuiltinImporter'>,
'__builtins__': <module 'builtins'>}
>>> animal
'fruitbat'
```

函数 change_local() 的局部命名空间只含有局部变量 animal。全局命名空间含有全局变量 animal 以及其他一些东西。

名称中_和__的用法

以两个下划线 __ 开头和结束的名称都是 Python 的保留用法。因此，在自定义的变量中不能使用它们。选择这种命名模式是考虑到开发者一般是不会选择它们作为自己的变量的。

例如，一个函数的名称是系统变量 function.__name__，它的文档字符串是 function.__doc__：

```
>>> def amazing():
...     '''This is the amazing function.
...     Want to see it again?'''
...     print('This function is named:', amazing.__name__)
...     print('And its docstring is:', amazing.__doc__)
...
>>> amazing()
This function is named: amazing
And its docstring is: This is the amazing function.
    Want to see it again?
```

如同之前 globals 的输出结果所示，主程序被赋值特殊的名字 __main__。

4.11 使用try和except处理错误

> 要么做，要么不做，没有尝试这回事。[3]
>
> ——尤达（Yoda）

在一些编程语言中，错误是通过特殊的函数返回值指出的，而 Python 使用异常，它是一段

注 3："Do, or do not. There is no try."科幻电影《星球大战》中绝地大师尤达 Yoda 的台词。——译者注

只有错误发生时执行的代码。

之前已经接触到一些有关错误的例子，例如读取列表或者元组的越界位置或者字典中不存在的键。所以，当你执行可能出错的代码时，需要适当的*异常处理程序*用于阻止潜在的错误发生。

在异常可能发生的地方添加异常处理程序，对于用户明确错误是一种好方法。即使不会及时解决问题，至少会记录运行环境并且停止程序执行。如果发生在某些函数中的异常不能被立刻捕捉，它会持续，直到被某个调用函数的异常处理程序所捕捉。在你不能提供自己的异常捕获代码时，Python 会输出错误消息和关于错误发生处的信息，然后终止程序，例如下面的代码段：

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> short_list[position]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

与其让错误随机产生，不如使用 `try` 和 `except` 提供错误处理程序：

```
>>> short_list = [1, 2, 3]
>>> position = 5
>>> try:
...     short_list[position]
... except:
...     print('Need a position between 0 and', len(short_list)-1, ' but got',
...            position)
...
Need a position between 0 and 2 but got 5
```

在 `try` 中的代码块会被执行。如果存在错误，就会抛出异常，然后执行 `except` 中的代码；否则，跳过 `except` 块代码。

像前面那样指定一个无参数的 `except` 适用于任何异常类型。如果可能发生多种类型的异常，最好是分开进行异常处理。当然，没人强迫你这么做，你可以使一个 `except` 去捕捉所有的异常，但是这样的处理方式会比较泛化（类似于直接输出*发生了一个错误*）。当然也可以使用任意数量的异常处理程序。

有时需要除了异常类型以外其他的异常细节，可以使用下面的格式获取整个异常对象：

```
except exceptiontype as name
```

下面的例子首先会寻找是否有 `IndexError`，因为它是由索引一个序列的非法位置抛出的异常类型。将一个 `IndexError` 异常赋给变量 `err`，把其他的异常赋给变量 `other`。示例中会输出所有存储在 `other` 中的该对象的异常。

```
>>> short_list = [1, 2, 3]
>>> while True:
...     value = input('Position [q to quit]? ')
...     if value == 'q':
...         break
```

```
...     try:
...         position = int(value)
...         print(short_list[position])
...     except IndexError as err:
...         print('Bad index:', position)
...     except Exception as other:
...         print('Something else broke:', other)
...
Position [q to quit]? 1
2
Position [q to quit]? 0
1
Position [q to quit]? 2
3
Position [q to quit]? 3
Bad index: 3
Position [q to quit]? 2
3
Position [q to quit]? two
Something else broke: invalid literal for int() with base 10: 'two'
Position [q to quit]? q
```

输入 3 会抛出异常 `IndexError`；输入 two 会使函数 `int()` 抛出异常，被第二个 except 所捕获。

4.12 编写自己的异常

前面一节讨论了异常处理，但是其中讲到的所有异常（例如 `IndexError`）都是在 Python 或者它的标准库中提前定义好的。根据自己的目的可以使用任意的异常类型，同时也可以自己定义异常类型，用来处理程序中可能会出现的特殊情况。

> 这里需要定义一个类的新对象，这会在第 6 章深入说明。如果你对类不是很熟悉，可以学完后面的部分再返回这一节。

一个异常是一个类，即类 Exception 的一个子类。现在编写异常 `UppercaseException`，在一个字符串中碰到大写字母会被抛出。

```
>>> class UppercaseException(Exception):
...     pass
...
>>> words = ['eeenie', 'meenie', 'miny', 'MO']
>>> for word in words:
...     if word.isupper():
...         raise UppercaseException(word)
...
Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
__main__.UppercaseException: MO
```

即使没有定义 `UppercaseException` 的行为（注意到只使用 `pass`），也可以通过继承其父类

Exception 在抛出异常时输出错误提示。

你当然能够访问异常对象本身，并且输出它：

```
>>> try:
...     raise OopsException('panic')
... except OopsException as exc:
...     print(exc)
...
panic
```

4.13 练习

(1) 将 7 赋值给变量 guess_me，然后写一段条件判断（if、else 和 elif）的代码：如果 guess_me 小于 7 输出 'too low'；大于 7 则输出 'too high'；等于 7 则输出 'just right'。

(2) 将 7 赋值给变量 guess_me，再将 1 赋值给变量 start。写一段 while 循环代码，比较 start 和 guess_me：如果 start 小于 guess_me，输出 too low；如果等于则输出 'found it!' 并终止循环；如果大于则输出 'oops'，然后终止循环。在每次循环结束时自增 start。

(3) 使用 for 循环输出列表 [3, 2, 1, 0] 的值。

(4) 使用列表推导生成 0~9（range(10)）的偶数列表。

(5) 使用字典推导创建字典 squares，把 0~9（range(10)）的整数作为键，每个键的平方作为对应的值。

(6) 使用集合推导创建集合 odd，包含 0~9（range(10)）的奇数。

(7) 使用生成器推导返回字符串 'Got ' 和 0~9 内的一个整数，使用 for 循环进行迭代。

(8) 定义函数 good：返回列表 ['Harry','Ron','Hermione']。

(9) 定义一个生成器函数 get_odds：返回 0~9 内的奇数。使用 for 循环查找并输出返回的第三个值。

(10) 定义一个装饰器 test：当一个函数被调用时输出 'start'，当函数结束时输出 'end'。

(11) 定义一个异常 OopsException：编写代码捕捉该异常，并输出 'Caught an oops'。

(12) 使用函数 zip() 创建字典 movies：匹配两个列表 titles = ['Creature of Habit', 'Crewel Fate'] 和 plots = ['A nun turns into a monster', 'A haunted yarn shop']。

第 5 章

Python盒子：模块、包和程序

在自底向上的学习过程中，我们已经学完内置的数据类型，以及构建较大的数据和代码结构。本章落到实质问题，学习如何写出实用的大型 Python 程序。

5.1 独立的程序

到目前为止，我们已经学会在 Python 的交互式解释器中编写和运行类似下面的代码：

```
>>> print("This interactive snippet works.")
This interactive snippet works.
```

现在编写你的第一个独立程序。在你的计算机中，创建一个文件 test1.py，包含下面的单行 Python 代码：

```
print("This standalone program works!")
```

注意，代码中没有 >>> 提示符，只有一行 Python 代码，而且要保证在 `print` 之前没有缩进。

如果要在文本终端或者终端窗口运行 Python，需要键入 Python 程序名，后面跟上程序的文件名：

```
$ python test1.py
This standalone program works!
```

> 你可以把在本书中已经看到的终端可交互的代码片段保存到文件中，然后直接运行。如果剪切或者粘贴，不要忘记删除 >>> 提示符和 ...（包括最后的空格）。

5.2　命令行参数

在你的计算机中，创建文件 test2.py，包含下面两行：

```
import sys
print('Program arguments:',sys.argv)
```

现在，使用 Python 运行这段程序。下面是在 Linux 或者 Mac OS X 系统的标准 shell 程序下的运行结果：

```
$ python test2.py
Program arguments: ['test2.py']
$ python test2.py tra la la
Program arguments: ['test2.py', 'tra', 'la', 'la']
```

5.3　模块和import语句

继续进入下一个阶段：在多个文件之间创建和使用 Python 代码。一个模块仅仅是 Python 代码的一个文件。

本书的内容按照这样的层次组织：单词、句子、段落以及章。否则，超过一两页后就没有很好的可读性了。代码也有类似的自底向上的组织层次：数据类型类似于单词，语句类似于句子，函数类似于段落，模块类似于章。以此类推，当我说某个内容会在第 8 章中说明时，就像是在其他模块中引用代码。

引用其他模块的代码时使用 `import` 语句，被引用模块中的代码和变量对该程序可见。

5.3.1　导入模块

`import` 语句最简单的用法是 `import` 模块，模块是不带 .py 扩展的另外一个 Python 文件的文件名。现在来模拟一个气象站，并输出天气预报。其中一个主程序输出报告，一个单独的具有单个函数的模块返回天气的描述。

下面是主程序（命名为 weatherman.py）：

```
import report

description = report.get_description()
print("Today's weather:", description)
```

以下是天气模块的代码（report.py）：

```
def get_description():  #看到下面的文档字符串了吗?
    """Return random weather, just like the pros"""
    from random import choice
    possibilities = ['rain', 'snow', 'sleet', 'fog', 'sun', 'who knows']
    return choice(possibilities)
```

如果上述两个文件在同一个目录下，通过 Python 运行主程序 weatherman.py，会引用 `report` 模块，执行函数 `get_description()`。函数 `get_description()` 从字符串列表中返回

一个随机结果。下面就是主程序可能返回和输出的结果：

```
$ python weatherman.py
Today's weather: who knows
$ python weatherman.py
Today's weather: sun
$ python weatherman.py
Today's weather: sleet
```

我们在两个不同的地方使用了 `import`：

- 主程序 weatherman.py 导入模块 `report`；
- 在模块文件 report.py 中，函数 `get_description()` 从 Python 标准模块 `random` 导入函数 `choice`。

同样，我们以两种不同的方式使用了 `import`：

- 主程序调用 `import report`，然后运行 `report.get_description()`；
- report.py 中的 `get_description()` 函数调用 `from random import choice`，然后运行 `choice(possibilities)`。

第一种情况下，我们导入了整个 `report` 模块，但是需要把 `report.` 作为 `get_description()` 的前缀。在这个 `import` 语句之后，只要在名称前加 `report.`，report.py 的所有内容（代码和变量）就会对主程序可见。通过模块名称限定模块的内容，可以避免命名冲突。其他模块可能也有函数 `get_description()`，这样做不会被错误地调用。

第二种情况下，所有代码都在同一个函数下，并且没有其他名为 `choice` 的函数，所以我们直接从 `random` 模块导入函数 `choice()`。我们也可以编写类似于下面的函数，返回随机结果：

```
def get_description():
    import random
    possibilities = ['rain', 'snow', 'sleet', fog', 'sun', 'who knows']
    return random.choice(possibilities)
```

同编程的其他方面一样，选择你所能理解的最清晰的风格。符合模块规范的命名（`random.choice`）更安全，但输入量略大。

这些 `get_description()` 的例子介绍了各种各样的导入内容，但没有涉及在什么地方进行导入——它们都在函数内部调用 `import`。我们也可以在函数外部导入 `random`：

```
>>> import random
>>> def get_description():
...     possibilities = ['rain', 'snow', 'sleet', 'fog', 'sun', 'who knows']
...     return random.choice(possibilities)
...
>>> get_description()
'who knows'
>>> get_description()
'rain'
```

如果被导入的代码被多次使用，就应该考虑在函数外部导入；如果被导入的代码使用有

限，就在函数内部导入。一些人更喜欢把所有的 `import` 都放在文件的开头，从而使代码之间的依赖关系清晰。两种方法都是可行的。

5.3.2　使用别名导入模块

在主程序 weatherman.py 中，我们调用了 `import report`。但是，如果存在同名的另一个模块或者你想使用更短更好记的名字，该如何做呢？在这种情况下，可以使用*别名* `wr` 进行导入：

```
import report as wr
description = wr.get_description()
print("Today's weather:", description)
```

5.3.3　导入模块的一部分

在 Python 中，可以导入一个模块的若干部分。每一部分都有自己的原始名字或者你起的别名。首先，从 `report` 模块中用原始名字导入函数 `get_description()`：

```
from report import get_description
description = get_description()
print("Today's weather:", description)
```

用它的别名 `do_it` 导入：

```
from report import get_description as do_it
description = do_it()
print("Today's weather:", description)
```

5.3.4　模块搜索路径

Python 会在什么地方寻找文件来导入模块？使用命名为 `path` 变量的存储在标准 `sys` 模块下的一系列目录名和 ZIP 压缩文件。你可以读取和修改这个列表。下面是在我的 Mac 上 Python 3.3 的 `sys.path` 的内容：

```
>>> import sys
>>> for place in sys.path:
...         print(place)
...

/Library/Frameworks/Python.framework/Versions/3.3/lib/python33.zip
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/plat-darwin
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/lib-dynload
/Library/Frameworks/Python.framework/Versions/3.3/lib/python3.3/site-packages
```

最开始的空白输出行是空字符串 `''`，代表当前目录。如果空字符串是在 `sys.path` 的开始位置，Python 会先搜索当前目录：`import report` 会寻找文件 report.py。

第一个匹配到的模块会先被使用，这也就意味着如果你在标准库之前的搜索路径上定义一个模块 `random`，就不会导入标准库中的 `random` 模块。

5.4 包

我们已使用过单行代码、多行函数、独立程序以及同一目录下的多个模块。为了使 Python 应用更具可扩展性，你可以把多个模块组织成文件层次，称之为包。

也许我们需要两种类型的天气预报：一种是次日的，一种是下周的。一种可行的方式是新建目录 sources，在该目录中新建两个模块 daily.py 和 weekly.py。每一个模块都有一个函数 `forecast`。每天的版本返回一个字符串，每周的版本返回包含 7 个字符串的列表。

下面是主程序和两个模块（函数 `enumerate()` 拆分一个列表，并对列表中的每一项通过 `for` 循环增加数字下标）。

主程序是 boxes/weather.py：

```
from sources import daily, weekly

print("Daily forecast:", daily.forecast())
print("Weekly forecast:")
for number, outlook in enumerate(weekly.forecast(), 1):
    print(number, outlook)
```

模块 1 是 boxes/sources/daily.py：

```
def forecast():
    'fake daily forecast'
    return 'like yesterday'
```

模块 2 是 boxes/sources/weekly.py：

```
def forecast():
    """Fake weekly forecast"""
    return ['snow', 'more snow', 'sleet',
        'freezing rain', 'rain', 'fog', 'hail']
```

还需要在 sources 目录下添加一个文件：init.py。这个文件可以是空的，但是 Python 需要它，以便把该目录作为一个包。

运行主程序 weather.py：

```
$ python weather.py
Daily forecast: like yesterday
Weekly forecast:
1 snow
2 more snow
3 sleet
4 freezing rain
5 rain
6 fog
7 hail
```

5.5 Python标准库

Python 的一个显著特点是具有庞大的模块标准库，这些模块可以执行很多有用的任务，并

且和核心 Python 语言分开以避免臃肿。当我们开始写代码时，首先要检查是否存在想要的标准模块。在标准库中你会经常碰到一些“珍宝”！ Python 同时提供了模块的官网文档（http://docs.python.org/3/library）以及使用指南（https://docs.python.org/3.3/tutorial/stdlib.html）。Doug Hellmann 的 网 站 Python Module of the Week（http://pymotw.com/2/contents.html）和他的书 *The Python Standard Library by Example* 都是非常有帮助的指南。

接下来的几章会着重介绍关于网络、系统、数据库等的标准模块。本节讨论一些常用的标准模块。

5.5.1　使用setdefault()和defaultdict()处理缺失的键

读取字典中不存在的键的值会抛出异常。使用字典函数 `get()` 返回一个默认值会避免异常发生。函数 `setdefault()` 类似于 `get()`，但当键不存在时它会在字典中添加一项：

```
>>> periodic_table = {'Hydrogen': 1, 'Helium': 2}
>>> print(periodic_table)
{'Helium': 2, 'Hydrogen': 1}
```

如果键不在字典中，新的默认值会被添加进去：

```
>>> carbon = periodic_table.setdefault('Carbon', 12)
>>> carbon
12
>>> periodic_table
{'Helium': 2, 'Carbon': 12, 'Hydrogen': 1}
```

如果试图把一个不同的默认值赋给已经存在的键，不会改变原来的值，仍将返回初始值：

```
>>> helium = periodic_table.setdefault('Helium', 947)
>>> helium
2
>>> periodic_table
{'Helium': 2, 'Carbon': 12, 'Hydrogen': 1}
```

`defaultdict()` 也有同样的用法，但是在创建字典时，对每个新的键都会指定默认值。它的参数是一个函数。在本例中，把函数 `int` 作为参数传入，会按照 `int()` 调用，返回整数 `0`：

```
>>> from collections import defaultdict
>>> periodic_table = defaultdict(int)
```

现在，任何缺失的值将被赋为整数 `0`：

```
>>> periodic_table['Hydrogen'] = 1
>>> periodic_table['Lead']
0
>>> periodic_table
defaultdict(<class 'int'>, {'Lead': 0, 'Hydrogen': 1})
```

函数 `defaultdict()` 的参数是一个函数，它返回赋给缺失键的值。在下面的例子中，`no_idea()` 在需要时会被执行，返回一个值：

```
>>> from collections import defaultdict
>>>
>>> def no_idea():
```

```
...     return 'Huh?'
...
>>> bestiary = defaultdict(no_idea)
>>> bestiary['A'] = 'Abominable Snowman'
>>> bestiary['B'] = 'Basilisk'
>>> bestiary['A']
'Abominable Snowman'
>>> bestiary['B']
'Basilisk'
>>> bestiary['C']
'Huh?'
```

同样，可以使用函数 int()、list() 或者 dict() 返回默认空的值：int() 返回 0，list() 返回空列表（[]），dict() 返回空字典（{}）。如果你删掉该函数参数，新键的初始值会被设置为 None。

顺便提一下，也可以使用 lambda 来定义你的默认值函数：

```
>>> bestiary = defaultdict(lambda: 'Huh?')
>>> bestiary['E']
'Huh?'
```

使用 int 是一种定义计数器的方式：

```
>>> from collections import defaultdict
>>> food_counter = defaultdict(int)
>>> for food in ['spam', 'spam', 'eggs', 'spam']:
...     food_counter[food] += 1
...
>>> for food, count in food_counter.items():
...     print(food, count)
...
eggs 1
spam 3
```

上面的例子中，如果 food_counter 已经是一个普通的字典而不是 defaultdict 默认字典，那每次试图自增字典元素 food_counter[food] 值时，Python 会抛出一个异常，因为我们没有对它进行初始化。在普通字典中，需要做额外的工作，如下所示：

```
>>> dict_counter = {}
>>> for food in ['spam', 'spam', 'eggs', 'spam']:
...     if not food in dict_counter:
...         dict_counter[food] = 0
...     dict_counter[food] += 1
...
>>> for food, count in dict_counter.items():
...     print(food, count)
...
spam 3
eggs 1
```

5.5.2　使用Counter()计数

说起计数器，标准库有一个计数器，它可以胜任之前或者更多示例所做的工作：

```
>>> from collections import Counter
>>> breakfast = ['spam', 'spam', 'eggs', 'spam']
>>> breakfast_counter = Counter(breakfast)
>>> breakfast_counter
Counter({'spam': 3, 'eggs': 1})
```

函数 most_common() 以降序返回所有元素，或者如果给定一个数字，会返回该数字前的的元素：

```
>>> breakfast_counter.most_common()
[('spam', 3), ('eggs', 1)]
>>> breakfast_counter.most_common(1)
[('spam', 3)]
```

也可以组合计数器。首先来看一下 breakfast_counter：

```
>>> breakfast_counter
>>> Counter({'spam': 3, 'eggs': 1})
```

这一次，新建一个列表 lunch 和一个计数器 lunch_counter：

```
>>> lunch = ['eggs', 'eggs', 'bacon']
>>> lunch_counter = Counter(lunch)
>>> lunch_counter
Counter({'eggs': 2, 'bacon': 1})
```

第一种组合计数器的方式是使用 +：

```
>>> breakfast_counter + lunch_counter
Counter({'spam': 3, 'eggs': 3, 'bacon': 1})
```

你也可能想到，从一个计数器去掉另一个，可以使用 -。什么是早餐有的而午餐没有的呢？

```
>>> breakfast_counter - lunch_counter
Counter({'spam': 3})
```

那么什么又是午餐有的而早餐没有的呢 ?

```
>>> lunch_counter - breakfast_counter
Counter({'bacon': 1, 'eggs': 1})
```

和第 4 章中的集合类似，可以使用交集运算符 & 得到二者共有的项：

```
>>> breakfast_counter & lunch_counter
Counter({'eggs': 1})
```

两者的交集通过取两者中的较小计数，得到共同元素 'eggs'。这合情合理：早餐仅提供一个鸡蛋，因此也是共有的计数。

最后，使用并集运算符 | 得到所有元素：

```
>>> breakfast_counter | lunch_counter
Counter({'spam': 3, 'eggs': 2, 'bacon': 1})
```

'eggs' 又是两者共有的项。不同于合并，并集没有把计数加起来，而是取其中较大的值。

5.5.3 使用有序字典OrderedDict()按键排序

在前面几章的代码示例中可以看出，一个字典中键的顺序是不可预知的：你可以按照顺序添加键 a、b 和 c，但函数 keys() 可能返回 c、a 和 b。下面是第 1 章用过的一个例子：

```
>>> quotes = {
...     'Moe': 'A wise guy, huh?',
...     'Larry': 'Ow!',
...     'Curly': 'Nyuk nyuk!',
...     }
>>> for stooge in quotes:
...  print(stooge)
...
Larry
Curly
Moe
```

有序字典 OrderedDict() 记忆字典键添加的顺序，然后从一个迭代器按照相同的顺序返回。试着用元组（键，值）创建一个有序字典：

```
>>> from collections import OrderedDict
>>> quotes = OrderedDict([
...     ('Moe', 'A wise guy, huh?'),
...     ('Larry', 'Ow!'),
...     ('Curly', 'Nyuk nyuk!'),
...     ])
>>>
>>> for stooge in quotes:
...     print(stooge)
...
Moe
Larry
Curly
```

5.5.4 双端队列：栈+队列

deque 是一种双端队列，同时具有栈和队列的特征。它可以从序列的任何一端添加和删除项。现在，我们从一个词的两端扫向中间，判断是否为回文。函数 popleft() 去掉最左边的项并返回该项，pop() 去掉最右边的项并返回该项。从两边一直向中间扫描，只要两端的字符匹配，一直弹出直到到达中间：

```
>>> def palindrome(word):
...     from collections import deque
...     dq = deque(word)
...     while len(dq) > 1:
...        if dq.popleft() != dq.pop():
...            return False
...     return True
...
...
>>> palindrome('a')
True
>>> palindrome('racecar')
```

```
True
>>> palindrome('')
True
>>> palindrome('radar')
True
>>> palindrome('halibut')
False
```

这里把判断回文作为双端队列的一个简单说明。如果想要写一个快速的判断回文的程序，只需要把字符串反转和原字符串进行比较。Python 没有对字符串进行反转的函数 `reverse()`，但还是可以利用反向切片的方式进行反转，如下所示：

```
>>> def another_palindrome(word):
...     return word == word[::-1]
...
>>> another_palindrome('radar')
True
>>> another_palindrome('halibut')
False
```

5.5.5 使用itertools迭代代码结构

`itertools`（https://docs.python.org/3/library/itertools.html）包含特殊用途的迭代器函数。在 `for ... in` 循环中调用迭代函数，每次会返回一项，并记住当前调用的状态。

即使 `chain()` 的参数只是单个迭代对象，它也会使用参数进行迭代：

```
>>> import itertools
>>> for item in itertools.chain([1, 2], ['a', 'b']):
...     print(item)
...
1
2
a
b
```

`cycle()` 是一个在它的参数之间循环的无限迭代器：

```
>>> import itertools
>>> for item in itertools.cycle([1, 2]):
...     print(item)
...
1
2
1
2
.
.
.
```

`accumulate()` 计算累积的值。默认的话，它计算的是累加和：

```
>>> import itertools
>>> for item in itertools.accumulate([1, 2, 3, 4]):
...     print(item)
```

```
...
1
3
6
10
```

你可以把一个函数作为 `accumulate()` 的第二个参数，代替默认的加法函数。这个参数函数应该接受两个参数，返回单个结果。下面的例子计算的是乘积：

```
>>> import itertools
>>> def multiply(a, b):
...     return a * b
...
>>> for item in itertools.accumulate([1, 2, 3, 4], multiply):
...     print(item)
...
1
2
6
24
```

`itertools` 模块有很多其他的函数，有一些可以用在需要节省时间的组合和排列问题上。

5.5.6　使用pprint()友好输出

我们见到的所有示例都用 `print()`（或者在交互式解释器中用变量名）打印输出。有时输出结果的可读性较差。我们需要一个友好输出函数，比如 `pprint()`：

```
>>> from pprint import pprint
>>> quotes = OrderedDict([
...     ('Moe', 'A wise guy, huh?'),
...     ('Larry', 'Ow!'),
...     ('Curly', 'Nyuk nyuk!'),
...     ])
>>>
```

普通的 `print()` 直接列出所有结果：

```
>>> print(quotes)
OrderedDict([('Moe', 'A wise guy, huh?'), ('Larry', 'Ow!'), ('Curly', 'Nyuk nyuk!')])
```

但是，`pprint()` 尽量排列输出元素从而增加可读性：

```
>>> pprint(quotes)
{'Moe': 'A wise guy, huh?',
 'Larry': 'Ow!',
 'Curly': 'Nyuk nyuk!'}
```

5.6　获取更多Python代码

有时标准库没有你需要的代码，或者不能很好地满足需求。有很多开源第三方 Python 软件供参考，如下所示：

- PyPi（http://pypi.python.org，也称为 Cheese Shop，名称源自 Monty Python[1] 的滑稽短剧）
- github（http://github.com/Python）
- readthedocs（https://readthedocs.org/）

你可以在 activestate（http://code.activestate.com/recipes/langs/python/）找到很多小代码示例。

本书的绝大部分代码使用的是安装在你电脑中的标准 Python 程序，包括所有内置函数和标准库。外部的包在以下地方提及：第 1 章中谈到的 `requests`，具体细节在 9.1.3 节；附录 D 里面提及第三方 Python 软件的安装和详细的开发细节。

5.7　练习

(1) 创建文件 zoo.py。在该文件中定义函数 `hours()`，输出字符串 `'Open 9-5 daily'`。然后使用交互式解释器导入模块 `zoo` 并调用函数 `hours()`。
(2) 在交互式解释器中，把模块 `zoo` 作为 `menagerie` 导入，然后调用函数 `hours()`。
(3) 依旧在解释器中，直接从模块 `zoo` 导入函数 `hours()` 并调用。
(4) 把函数 `hours()` 作为 `info` 导入，然后调用它。
(5) 创建字典 `plain`，包含键值对 `'a':1`、`'b':2` 和 `'c':3`，然后输出它。
(6) 创建有序字典 `fancy`：键值对和练习 (5) 相同，然后输出它。输出顺序和 `plain` 相同吗？
(7) 创建默认字典 `dict_of_lists`，传入参数 `list`。给 `dict_of_lists['a']` 赋值 `'something for a'`，输出 `dict_of_lists['a']` 的值。

注 1：Monty Python 是英国六人喜剧团体。——译者注

第 6 章

对象和类

“对象并不神秘，神秘的是你的眼睛。”[1]

——Elizabeth Bowen

“选取一个对象，对它进行修改，然后再进行一些其他的修改。”[2]

——Jasper Johns

到目前为止，你已经学习了字符串、字典之类的数据结构，还有函数、模块之类的代码结构。本章将学习如何使用自定义的数据结构：对象。

6.1　什么是对象

就像在第 2 章提到的一样，Python 里的所有数据都是以对象形式存在的，无论是简单的数字类型还是复杂的代码模块。然而，Python 特殊的语法形式巧妙地将实现对象机制的大量细节隐藏了起来。输入 `num = 7` 就可以创建一个值为 7 的整数对象，并且将这个对象赋值给变量 `num`。事实上，在 Python 中，只有当你想要创建属于自己的对象或者需要修改已有对象的行为时，才需要关注对象的内部实现细节。在本章，这两种情况都会涉及。

对象既包含数据（变量，更习惯称之为特性，attribute），也包含代码（函数，也称为方法）。它是某一类具体事物的特殊实例。例如，整数 7 就是一个包含了加法、乘法之类方法的对象，这些方法在 2.2 节曾经介绍过。整数 8 则是另一个对象。这意味着在 Python 里，7 和 8

注 1：“No object is mysterious. The mystery is your eye.”原话应译为：没有什么是神秘的，神秘的是你的眼睛。本书作者使用 object 一语双关。——译者注

注 2：“Take an object. Do something to it. Do somthing else to it.”原话应译为：选取一件事物，对它进行创作，再对它进行其他创作。本书作者使用 object 一语双关。——译者注

都属于一个公共的类，我们称之为整数类。字符串 'cat' 和 'duck' 也是 Python 对象，它们都包含着 capitalize() 和 replace() 之类的字符串方法，这些方法之前也都见到过。

当你想要创建一个别人从来没有创建过的新对象时，首先必须定义一个类，用以指明该类型的对象所包含的内容（特性和方法）。

可以把对象想象成名词，那么方法就是动词。对象代表着一个独立的事物，它的方法则定义了它是如何与其他事物相互作用的。

与模块不同，你可以同时创建许多同类的对象，它们的特性值可能各不相同。对象就像是包含了代码的超级数据结构。

6.2 使用class定义类

在第 1 章，我把对象比作塑料盒子。类（class）则像是制作盒子用的模具。例如，Python 的内置类 String 可以创建像 'cat' 和 'duck' 这样的字符串对象。Python 中还有许多用来创建其他标准数据类型的类，包括列表、字典等。如果想要在 Python 中创建属于自己的对象，首先你必须用关键词 class 来定义一个类。先来看一个简单的例子。

假设你想要定义一些对象用于记录联系人，每个对象对应一个人。首先需要定义 Person 类作为生产对象的模具。在接下来的几个例子中，我们会不停更新这个类的内容，从最简单的开始，直到它成为一个可以实际使用的类。

首先创建的是最简单的类，即一个没有任何内容的空类：

```
>>> class Person():
...     pass
```

同函数一样，用 pass 表示这个类是一个空类。上面这种定义类的方法已经是最简形式，无法再省略。你可以通过类名来创建对象，同调用函数一样：

```
>>> someone = Person()
```

在这个例子中，Person() 创建了一个 Person 类的对象，并给它赋值 someone 这个名字。但是，由于我们的 Person 类是空的，所以由它创建的对象 someone 实际上什么也做不了。实际编程中，你永远也不会创建这样一个没用的类，我在这里只是为了从零开始引出后面每一步的内容。

我们来试着重新定义一下 Person 类。这一次，将 Python 中特殊的对象初始化方法 __init__ 放入其中：

```
>>> class Person():
...     def __init__(self):
...         pass
```

我承认 __init__() 和 self 看起来很奇怪，但这就是实际的 Python 类的定义形式。__init__() 是 Python 中一个特殊的函数名，用于根据类的定义创建实例对象。[3]self 参数指

注 3：你会在 Python 中见到许多双下划线（double underscore）的名字。一些懒人喜欢将其简称为 dunder，这样比较节省音节。

向了这个正在被创建的对象本身。

当你在类声明里定义 `__init__()` 方法时，第一个参数必须为 `self`。尽管 `self` 并不是一个 Python 保留字，但它很常用。没有人（包括你自己）在阅读你的代码时需要猜测使用 `self` 的意图。

尽管我们添加了初始化方法，但用这个 `Person` 类创建的对象仍然什么也做不了。即将进行的第三次尝试要更吸引人了，你将学习如何创建一个简单可用的 Python 对象。这一次，会在初始化方法中添加 `name` 参数：

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>>
```

现在，用 `Person` 类创建一个对象，为 `name` 特性传递一个字符串参数：

```
>>> hunter = Person('Elmer Fudd')
```

上面这短短的一行代码实际做了以下工作：

- 查看 `Person` 类的定义；
- 在内存中实例化（创建）一个新的对象；
- 调用对象的 `__init__` 方法，将这个新创建的对象作为 `self` 传入，并将另一个参数（`'Elmer-Fudd'`）作为 `name` 传入；
- 将 `name` 的值存入对象；
- 返回这个新的对象；
- 将名字 `hunter` 与这个对象关联。

这个新对象与任何其他的 Python 对象一样。你可以把它当作列表、元组、字典或集合中的元素，也可以把它当作参数传递给函数，或者把它做为函数的返回结果。

我们刚刚传入的 `name` 参数此时又在哪儿呢？它作为对象的特性存储在了对象里。可以直接对它进行读写操作：

```
>>> print('The mighty hunter: ', hunter.name)
The mighty hunter: Elmer Fudd
```

记住，在 `Person` 类定义的内部，你可以直接通过 `self.name` 访问 `name` 特性。而当创建了一个实际的对象后，例如这里的 `hunter`，需要通过 `hunter.name` 来访问它。

在类的定义中，`__init__` 并不是必需的。只有当需要区分由该类创建的不同对象时，才需要指定 `__init__` 方法。

6.3 继承

在你编写代码解决实际问题时，经常能找到一些已有的类，它们能够实现你所需的大部分功能，但不是全部。这时该怎么办？当然，你可以对这个已有的类进行修改，但这么做很

容易让代码变得更加复杂，一不留神就可能会破坏原来可以正常工作的功能。

当然，也可以另起炉灶重新编写一个类：复制粘贴原来的代码再融入自己的新代码。但这意味着你需要维护更多的代码。同时，新类和旧类中实现同样功能的代码被分隔在了不同的地方（日后修改时需要改动多处）。

更好的解决方法是利用类的*继承*：从已有类中衍生出新的类，添加或修改部分功能。这是代码复用的一个绝佳的例子。使用继承得到的新类会自动获得旧类中的所有方法，而不需要进行任何复制。

你只需要在新类里面定义自己额外需要的方法，或者按照需求对继承的方法进行修改即可。修改得到的新方法会覆盖原有的方法。我们习惯将原始的类称为*父类*、*超类*或*基类*，将新的类称作*孩子类*、*子类*或*衍生类*。这些术语在面向对象的编程中不加以区分。

现在，我们来试试继承。首先，定义一个空类 Car。然后，定义一个 Car 的子类 Yugo。定义子类使用的也是 class 关键词，不过需要把父类的名字放在子类名字后面的括号里（class Yugo(Car)）：

```
>>> class Car():
...     pass
...
>>> class Yugo(Car):
...     pass
...
```

接着，为每个类创建一个实例对象：

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

子类是父类的一种特殊情况，它属于父类。在面向对象的术语里，我们经常称 Yugo 是一个（is-a）Car。对象 give_me_a_yugo 是 Yugo 类的一个实例，但它同时继承了 Car 能做到的所有事情。当然，上面的例子中 Car 和 Yugo 就像潜艇上的甲板水手一样起不到任何实际作用。我们来更新一下类的定义，让它们发挥点儿作用：

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     pass
...
```

最后，为每一个类各创建一个对象，并调用刚刚声明的 exclaim 方法：

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Car!
```

我们不需要进行任何特殊的操作，Yugo 就自动从 Car 那里继承了 exclaim() 方法。但事

实上，我们并不希望 Yugo 在 exlaim() 方法里宣称它是一个 Car，这可能会造成身份危机（无法区分 Car 和 Yugo）。让我们来看看怎么解决这个问题。

6.4　覆盖方法

就像上面的例子展示的一样，新创建的子类会自动继承父类的所有信息。接下来将看到子类如何替代——更习惯说*覆盖*（override）——父类的方法。Yugo 和 Car 一定存在着某些区别，不然的话，创建它又有什么意义？试着改写一下 Yugo 中 exclaim() 方法的功能：

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...
```

现在，为每个类创建一个对象：

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

看看它们各自会宣称什么？

```
>>> give_me_a_car.exclaim()
I'm a Car!
>>> give_me_a_yugo.exclaim()
I'm a Yugo! Much like a Car, but more Yugo-ish.
```

在上面的例子中，我们覆盖了父类的 exclaim() 方法。在子类中，可以覆盖任何父类的方法，包括 __init__()。下面的例子使用了之前创建过的 Person 类。我们来创建两个子类，分别代表医生（MDPerson）和律师（JDPerson）：

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
>>> class MDPerson(Person):
...     def __init__(self, name):
...         self.name = "Doctor " + name
...
>>> class JDPerson(Person):
...     def __init__(self, name):
...         self.name = name + ", Esquire"
...
```

在上面的例子中，子类的初始化方法 __init__() 接收的参数和父类 Person 一样，但存储到对象内部 name 特性的值却不尽相同：

```
>>> person = Person('Fudd')
>>> doctor = MDPerson('Fudd')
>>> lawyer = JDPerson('Fudd')
```

```
>>> print(person.name)
Fudd
>>> print(doctor.name)
Doctor Fudd
>>> print(lawyer.name)
Fudd, Esquire
```

6.5 添加新方法

子类还可以添加父类中没有的方法。回到 Car 类和 Yugo 类，我们给 Yugo 类添加一个新的方法 need_a_push()：

```
>>> class Car():
...     def exclaim(self):
...         print("I'm a Car!")
...
>>> class Yugo(Car):
...     def exclaim(self):
...         print("I'm a Yugo! Much like a Car, but more Yugo-ish.")
...     def need_a_push(self):
...         print("A little help here?")
...
```

接着，创建一个 Car 和一个 Yugo 对象：

```
>>> give_me_a_car = Car()
>>> give_me_a_yugo = Yugo()
```

Yugo 类的对象可以响应 need_a_push() 方法：

```
>>> give_me_a_yugo.need_a_push()
A little help here?
```

但比它广义的 Car 无法响应该方法：

```
>>> give_me_a_car.need_a_push()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Car' object has no attribute 'need_a_push'
```

至此，Yugo 终于可以做一些 Car 做不到的事情了。它的与众不同的特征开始体现了出来。

6.6 使用super从父类得到帮助

我们已经知道如何在子类中覆盖父类的方法，但如果想要调用父类的方法怎么办？“哈哈！终于等到你问这个了。”super() 站出来说道。下面的例子将定义一个新的类 EmailPerson，用于表示有电子邮箱的 Person。首先，来定义熟悉的 Person 类：

```
>>> class Person():
...     def __init__(self, name):
...         self.name = name
...
```

下面是子类的定义。注意，子类的初始化方法 __init__() 中添加了一个额外的 email 参数：

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         super().__init__(name)
...         self.email = email
```

在子类中定义 __init__() 方法时，父类的 __init__() 方法会被覆盖。因此，在子类中，父类的初始化方法并不会被自动调用，我们必须显式调用它。以上代码实际上做了这样几件事情。

- 通过 super() 方法获取了父类 Person 的定义。
- 子类的 __init__() 调用了 Person.__init__() 方法。它会自动将 self 参数传递给父类。因此，你只需传入其余参数即可。在上面的例子中，Person() 能接受的其余参数指的是 name。
- self.email = email 这行新的代码才真正起到了将 EmailPerson 与 Person 区分开的作用。

接下来，创建一个 EmailPerson 类的对象：

```
>>> bob = EmailPerson('Bob Frapples', 'bob@frapples.com')
```

我们既可以访问 name 特性，也可以访问 email 特性：

```
>>> bob.name
'Bob Frapples'
>>> bob.email
'bob@frapples.com'
```

为什么不像下面这样定义 EmailPerson 类呢？

```
>>> class EmailPerson(Person):
...     def __init__(self, name, email):
...         self.name = name
...         self.email = email
```

确实可以这么做，但这有悖我们使用继承的初衷。我们应该使用 super() 来让 Person 完成它应该做的事情，就像任何一个单纯的 Person 对象一样。除此之外，不这么写还有另一个好处：如果 Person 类的定义在未来发生改变，使用 super() 可以保证这些改变会自动反映到 EmailPerson 类上，而不需要手动修改。

子类可以按照自己的方式处理问题，但如果仍需要借助父类的帮助，使用 super() 是最佳的选择（就像现实生活中孩子与父母的关系一样）。

6.7 self的自辩

Python 中经常被争议的一点（除了使用空格[4]外）就是必须把 self 设置为实例方法（前面

注 4：Python 使用空格而不使用 tab 进行缩进，并且用缩进标志代码块，这使得不同开发者合作时必须事先统一缩进空格的数量，不然会造成代码混乱。这是 Python 被争议最多的毛病之一。——译者注

例子中你见到的所有方法都是实例方法）的第一个参数。Python 使用 `self` 参数来找到正确的对象所包含的特性和方法。通过下面的例子，我会告诉你调用对象方法背后 Python 实际做的工作。

还记得前面例子中的 `Car` 类吗？再次调用 `exclaim()` 方法：

```
>>> car = Car()
>>> car.exclaim()
I'm a Car!
```

Python 在背后做了以下两件事情：

- 查找 `car` 对象所属的类（`Car`）；
- 把 `car` 对象作为 `self` 参数传给 `Car` 类所包含的 `exclaim()` 方法。

了解调用机制后，为了好玩，我们甚至可以像下面这样进行调用，这与普通的调用语法（`car.exclaim()`）效果完全一致：

```
>>> Car.exclaim(car)
I'm a Car!
```

当然，我们没有理由使用这种臃肿的语法。

6.8　使用属性对特性进行访问和设置

有一些面向对象的语言支持私有特性。这些特性无法从对象外部直接访问，我们需要编写 getter 和 setter 方法对这些私有特性进行读写操作。

Python 不需要 getter 和 setter 方法，因为 Python 里所有特性都是公开的，使用时全凭自觉。如果你不放心直接访问对象的特性，可以为对象编写 setter 和 getter 方法。但更具 Python 风格的解决方案是使用*属性*（property）[5]。

下面的例子中，首先定义一个 `Duck` 类，它仅包含一个 `hidden_name` 特性。（下一节会告诉你命名私有特性的一种更好的方式。）我们不希望别人能够直接访问这个特性，因此需要定义两个方法：getter 方法（`get_name()`）和 setter 方法（`set_name()`）。我们在每个方法中都添加一个 `print()` 函数，这样就能方便地知道它们何时被调用。最后，把这些方法设置为 `name` 属性：

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     def get_name(self):
...         print('inside the getter')
...         return self.hidden_name
...     def set_name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
...     name = property(get_name, set_name)
```

注 5：本书将 property 译作*属性*，而将 attribute 译作*特性*，请读者注意区分。——译者注

这两个新方法在最后一行之前都与普通的 getter 和 setter 方法没有任何区别，而最后一行则把这两个方法定义为了 name 属性。property() 的第一个参数是 getter 方法，第二个参数是 setter 方法。现在，当你尝试访问 Duck 类对象的 name 特性时，get_name() 会被自动调用：

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
```

当然，也可以显式调用 get_name() 方法，它就像普通的 getter 方法一样：

```
>>> fowl.get_name()
inside the getter
'Howard'
```

当对 name 特性执行赋值操作时，set_name() 方法会被调用：

```
>>> fowl.name = 'Daffy'
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

也可以显式调用 set_name() 方法：

```
>>> fowl.set_name('Daffy')
inside the setter
>>> fowl.name
inside the getter
'Daffy'
```

另一种定义属性的方式是使用修饰符（decorator）。下一个例子会定义两个不同的方法，它们都叫 name()，但包含不同的修饰符：

- @property，用于指示 getter 方法；
- @name.setter，用于指示 setter 方法。

实际代码如下所示：

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.hidden_name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.hidden_name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.hidden_name = input_name
```

你仍然可以像之前访问特性一样访问 name，但这里没有了显式的 get_name() 和 set_name() 方法：

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

实际上，如果有人能猜到我们在类的内部用的特性名是 hidden_name，他仍然可以直接通过 fowl.hidden_name 进行读写操作。下一节将看到 Python 中特有的命名私有特性的方式。

在前面几个例子中，我们都使用 name 属性指向类中存储的某一特性（在我们的例子中是 hidden_name）。除此之外，属性还可以指向一个计算结果值。我们来定义一个 Circle 类，它包含 radius 特性以及一个计算属性 diameter：

```
>>> class Circle():
...     def __init__(self, radius):
...         self.radius = radius
...     @property
...     def diameter(self):
...         return 2 * self.radius
...
```

创建一个 Circle 对象，并给 radius 赋予一个初值：

```
>>> c = Circle(5)
>>> c.radius
5
```

可以像访问特性（例如 radius）一样访问属性 diameter：

```
>>> c.diameter
10
```

真正有趣的还在后面。我们可以随时改变 radius 特性的值，计算属性 diameter 会自动根据新的值更新自己：

```
>>> c.radius = 7
>>> c.diameter
14
```

如果你没有指定某一特性的 setter 属性（@diameter.setter），那么将无法从类的外部对它的值进行设置。这对于那些只读的特性非常有用：

```
>>> c.diameter = 20
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: can't set attribute
```

与直接访问特性相比，使用 property 还有一个巨大的优势：如果你改变了某个特性的定义，只需要在类定义里修改相关代码即可，不需要在每一处调用修改。[6]

6.9　使用名称重整保护私有特性

前面的 Duck 例子中，为了隐藏内部特性，我们曾将其命名为 hidden_name。其实，Python 对那些需要刻意隐藏在类内部的特性有自己的命名规范：由连续的两个下划线开头（__）。

我们来把 hidden_name 改名为 __name，如下所示：

```
>>> class Duck():
...     def __init__(self, input_name):
...         self.__name = input_name
...     @property
...     def name(self):
...         print('inside the getter')
...         return self.__name
...     @name.setter
...     def name(self, input_name):
...         print('inside the setter')
...         self.__name = input_name
...
```

看看代码是否还能正常工作：

```
>>> fowl = Duck('Howard')
>>> fowl.name
inside the getter
'Howard'
>>> fowl.name = 'Donald'
inside the setter
>>> fowl.name
inside the getter
'Donald'
```

看起来不错！现在，你无法在外部访问 __name 特性了：

```
>>> fowl.__name
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Duck' object has no attribute '__name'
```

这种命名规范本质上并没有把特性变成私有，但 Python 确实将它的名字重整了，让外部的代码无法使用。如果你实在好奇名称重整是怎么实现的，我可以偷偷地告诉你其中的奥秘，但不要告诉别人哦：

```
>>> fowl._Duck__name
'Donald'
```

注 6：如前面例子中，假如我们需要把特性 hidden_name 的名字改成 in_class_name。不设置属性（property）的话，我们需要在每一处访问 hidden_name 的地方将它替换成 in_class_name；而设置了属性的话，仅需在类的内部修改，其余部分的访问仍直接通过属性 name 即可。——译者注

发现了吗？我们并没有得到 `inside the getter`，成功绕过了 getter 方法。尽管如我们所见，这种保护特性的方式并不完美，但它确实能在一定程度上避免我们无意或有意地对特性进行直接访问。

6.10 方法的类型

有些数据（特性）和函数（方法）是类本身的一部分，还有一些是由类创建的实例的一部分。

在类的定义中，以 `self` 作为第一个参数的方法都是*实例方法*（instance method）。它们在创建自定义类时最常用。实例方法的首个参数是 `self`，当它被调用时，Python 会把调用该方法的对象作为 `self` 参数传入。

与之相对，*类方法*（class method）会作用于整个类，对类作出的任何改变会对它的所有实例对象产生影响。在类定义内部，用前缀修饰符 `@classmethod` 指定的方法都是类方法。与实例方法类似，类方法的第一个参数是类本身。在 Python 中，这个参数常被写作 `cls`，因为全称 `class` 是保留字，在这里我们无法使用。下面的例子中，我们为类 `A` 定义一个类方法来记录一共有多少个类 `A` 的对象被创建：

```
>>> class A():
...     count = 0
...     def __init__(self):
...         A.count += 1
...     def exclaim(self):
...         print("I'm an A!")
...     @classmethod
...     def kids(cls):
...         print("A has", cls.count, "little objects.")
...
>>>
>>> easy_a = A()
>>> breezy_a = A()
>>> wheezy_a = A()
>>> A.kids()
A has 3 little objects.
```

注意，上面的代码中，我们使用的是 `A.count`（类特性），而不是 `self.count`（可能是对象的特性）。在 `kids()` 方法中，我们使用的是 `cls.count`，它与 `A.count` 的作用一样。

类定义中的方法还存在着第三种类型，它既不会影响类也不会影响类的对象。它们出现在类的定义中仅仅是为了方便，否则它们只能孤零零地出现在代码的其他地方，这会影响代码的逻辑性。这种类型的方法被称作*静态方法*（static method），用 `@staticmethod` 修饰，它既不需要 `self` 参数也不需要 `class` 参数。下面例子中的静态方法是一则 `CoyoteWeapon` 的广告：

```
>>> class CoyoteWeapon():
...     @staticmethod
...     def commercial():
...         print('This CoyoteWeapon has been brought to you by Acme')
...
```

```
>>>
>>> CoyoteWeapon.commercial()
This CoyoteWeapon has been brought to you by Acme
```

注意，在这个例子中，我们甚至都不用创建任何 CoyoteWeapon 类的对象就可以调用这个方法，句法优雅不失风格！

6.11 鸭子类型

Python 对实现多态（polymorphism）要求得十分宽松，这意味着我们可以对不同对象调用同名的操作，甚至不用管这些对象的类型是什么。

我们来为三个 Quote 类设定同样的初始化方法 __init__()，然后再添加两个新函数：

- who() 返回保存的 person 字符串的值；
- says() 返回保存的 words 字符串的内容，并添上指定的标点符号。

它们的具体实现如下所示：

```
>>> class Quote():
...     def __init__(self, person, words):
...         self.person = person
...         self.words = words
...     def who(self):
...         return self.person
...     def says(self):
...         return self.words + '.'
...
>>> class QuestionQuote(Quote):
...      def says(self):
...          return self.words + '?'
...
>>> class ExclamationQuote(Quote):
...      def says(self):
...          return self.words + '!'
...
>>>
```

我们不需要改变 QuestionQuote 或者 ExclamationQuote 的初始化方式，因此没有覆盖它们的 __init__() 方法。Python 会自动调用父类 Quote 的初始化函数 __init__() 来存储实例变量 person 和 words，这就是我们可以在子类 QuestionQuote 和 ExclamationQuote 的对象里访问 self.words 的原因。

接下来创建一些对象：

```
>>> hunter = Quote('Elmer Fudd', "I'm hunting wabbits")
>>> print(hunter.who(), 'says:', hunter.says())
Elmer Fudd says: I'm hunting wabbits.

>>> hunted1 = QuestionQuote('Bugs Bunny', "What's up, doc")
>>> print(hunted1.who(), 'says:', hunted1.says())
Bugs Bunny says: What's up, doc?
```

```
>>> hunted2 = ExclamationQuote('Daffy Duck', "It's rabbit season")
>>> print(hunted2.who(), 'says:', hunted2.says())
Daffy Duck says: It's rabbit season!
```

三个不同版本的 says() 为上面三种类提供了不同的响应方式，这是面向对象的语言中多态的传统形式。Python 在这方面走得更远一些，无论对象的种类是什么，只要包含 who() 和 says()，你便可以调用它。我们再来定义一个 BabblingBrook 类，它与我们之前的猎人猎物（Quote 类的后代）什么的没有任何关系：

```
>>> class BabblingBrook():
...     def who(self):
...         return 'Brook'
...     def says(self):
...         return 'Babble'
...
>>> brook = BabblingBrook()
```

现在，对不同对象执行 who() 和 says() 方法，其中有一个（brook）与其他类型的对象毫无关联：

```
>>> def who_says(obj):
...     print(obj.who(), 'says', obj.says())
...
>>> who_says(hunter)
Elmer Fudd says I'm hunting wabbits.
>>> who_says(hunted1)
Bugs Bunny says What's up, doc?
>>> who_says(hunted2)
Daffy Duck says It's rabbit season!
>>> who_says(brook)
Brook says Babble
```

这种方式有时被称作*鸭子类型*（duck typing），这个命名源自一句名言：

> *如果它像鸭子一样走路，像鸭子一样叫，那么它就是一只鸭子。*[7]
>
> *—— 一位智者*

6.12 特殊方法

到目前为止，你已经能创建并使用基本对象了。现在再往深钻研一些。

当我们输入像 a = 3 + 8 这样的式子时，整数 3 和 8 是怎么知道如何实现 + 的？同样，a 又是怎么知道如何使用 = 来获取计算结果的？你可以使用 Python 的*特殊方法*（special method），有时也被称作*魔术方法*（magic method），来实现这些操作符的功能。别担心，不需要甘道夫[8] 的帮助，它们一点也不复杂。

注 7：源于“鸭子测试”。在鸭子类型中，并不关注对象本身的具体类型，只关注它能实现的功能。——译者注

注 8：英国作家 J. R. R. 托尔金小说《指环王》《霍比特人》中的巫师。——译者注

这些特殊方法的名称以双下划线（__）开头和结束。没错，你已经见过其中一个：__init__，它根据类的定义以及传入的参数对新创建的对象进行初始化。

假设你有一个简单的 Word 类，现在想要添加一个 equals() 方法来比较两个词是否一致，忽略大小写。也就是说，一个包含值 'ha' 的 Word 对象与包含 'HA' 的是相同的。

下面的代码是第一次尝试，创建一个普通方法 equals()。self.text 是当前 Word 对象所包含的字符串文本，equals() 方法将该字符串与 word2（另一个 Word 对象）所包含的字符串做比较：

```
>>> class Word():
...    def __init__(self, text):
...        self.text = text
...
...    def equals(self, word2):
...        return self.text.lower() == word2.text.lower()
...
```

接着创建三个包含不同字符串的 Word 对象：

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
```

当字符串 'ha' 和 'HA' 被转换为小写形式再进行比较时（我们就是这么做的），它们应该是相等的：

```
>>> first.equals(second)
True
```

但字符串 'eh' 无论如何与 'ha' 也不会相等：

```
>>> first.equals(third)
False
```

我们成功定义了 equals() 方法来进行小写转换并比较。但试想一下，如果能通过 if first == second 进行比较的话岂不更妙？这样类会更自然，表现得更像一个 Python 内置的类。好的，来试试吧，把前面例子中的 equals() 方法的名称改为 __eq__()（请先暂时接受，后面我会解释为什么这么命名）：

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...
```

修改就此结束，来看看新的版本能否正常工作：

```
>>> first = Word('ha')
>>> second = Word('HA')
>>> third = Word('eh')
>>> first == second
True
```

```
>>> first == third
False
```

太神奇了！是不是如同魔术一般？仅需将方法名改为 Python 里进行相等比较的特殊方法名 `__eq__()` 即可。表 6-1 和表 6-2 列出了最常用的一些魔术方法。

表6-1：和比较相关的魔术方法

方法名	使用
`__eq__(self, other)`	`self == other`
`__ne__(self, other)`	`self != other`
`__lt__(self, other)`	`self < other`
`__gt__(self, other)`	`self > other`
`__le__(self, other)`	`self <= other`
`__ge__(self, other)`	`self >= other`

表6-2：和数学相关的魔术方法

方法名	使用
`__add__(self, other)`	`self + other`
`__sub__(self, other)`	`self - other`
`__mul__(self, other)`	`self * other`
`__floordiv__(self, other)`	`self // other`
`__truediv__(self, other)`	`self / other`
`__mod__(self, other)`	`self % other`
`__pow__(self, other)`	`self ** other`

不仅数字类型可以使用像 +（魔术方法 `__add__()`）和 -（魔术方法 `__sub__()`）的数学运算符，一些其他的类型也可以使用。例如，Python 的字符串类型使用 + 进行拼接，使用 * 进行复制。关于字符串的魔术方法还有很多，你可以在 Python 3 在线文档的 Special method names（https://docs.python.org/3/reference/datamodel.html#special-method-names）里找到，其中最常用的一些参见下面的表 6-3。

表6-3：其他种类的魔术方法

方法名	使用
`__str__(self)`	`str(self)`
`__repr__(self)`	`repr(self)`
`__len__(self)`	`len(self)`

除了 `__init__()` 外，你会发现在编写类方法时最常用到的是 `__str__()`，它用于定义如何打印对象信息。`print()` 方法，`str()` 方法以及你将在第 7 章读到的关于字符串格式化的相关方法都会用到 `__str__()`。交互式解释器则用 `__repr__()` 方法输出变量。如果在你的类既没有定义 `__str__()` 也没有定义 `__repr__()`，Python 会输出类似下面这样的默认字符串：

```
 >>> first = Word('ha')
>>> first
```

```
<__main__.Word object at 0x1006ba3d0>
>>> print(first)
<__main__.Word object at 0x1006ba3d0>
```

我们将 `__str__()` 和 `__repr__()` 方法都添加到 `Word` 类里，让输出的对象信息变得更好看些：

```
>>> class Word():
...     def __init__(self, text):
...         self.text = text
...     def __eq__(self, word2):
...         return self.text.lower() == word2.text.lower()
...     def __str__(self):
...         return self.text
...     def __repr__(self):
...         return 'Word("'  self.text  '")'
...
>>> first = Word('ha')
>>> first          # uses __repr__
Word("ha")
>>> print(first)   # uses __str__
ha
```

更多关于魔术方法的内容请查看 Python 在线文档（https://docs.python.org/3/reference/datamodel.html#special-method-names）。

6.13 组合

如果你想要创建的子类在大多数情况下的行为都和父类相似的话（子类是父类的一种特殊情况，它们之间是 is-a 的关系），使用继承是非常不错的选择。建立复杂的继承关系确实很吸引人，但有些时候使用组合（composition）或聚合（aggregation）更加符合现实的逻辑（x 含有 y，它们之间是 has-a 的关系）。一只鸭子是鸟的一种（is-a），它有一条尾巴（has-a）。尾巴并不是鸭子的一种，它是鸭子的组成部分。下个例子中，我们会建立 `bill` 和 `tail` 对象，并将它们都提供给 `duck` 使用：

```
>>> class Bill():
...     def __init__(self, description):
...         self.description = description
...
>>> class Tail():
...     def __init__(self, length):
...         self.length = length
...
>>> class Duck():
...     def __init__(self, bill, tail):
...         self.bill = bill
...         self.tail = tail
...     def about(self):
...         print('This duck has a', bill.description, 'bill and a',
...               tail.length, 'tail')
...
>>> tail = Tail('long')
>>> bill = Bill('wide orange')
```

```
>>> duck = Duck(bill, tail)
>>> duck.about()
This duck has a wide orange bill and a long tail
```

6.14 何时使用类和对象而不是模块

有一些方法可以帮助你决定是把你的代码封装到类里还是模块里。

- 当你需要许多具有相似行为（方法）但不同状态（特性）的实例时，使用对象是最好的选择。
- 类支持继承，但模块不支持。
- 如果你想要保证实例的唯一性，使用模块是最好的选择。不管模块在程序中被引用多少次，始终只有一个实例被加载。（对 Java 和 C++ 程序员来说，如果读过 Erich Gamma 的《设计模式：可复用面向对象软件的基础》，可以把 Python 模块理解为单例。）
- 如果你有一系列包含多个值的变量，并且它们能作为参数传入不同的函数，那么最好将它们封装到类里面。举个例子，你可能会使用以 `size` 和 `color` 为键的字典代表一张彩色图片。你可以在程序中为每张图片创建不同的字典，并把它们作为参数传递给像 `scale()` 或者 `transform()` 之类的函数。但这么做的话，一旦你想要添加其他的键或者函数会变得非常麻烦。为了保证统一性，应该定义一个 `Image` 类，把 `size` 和 `color` 作为特性，把 `scale()` 和 `transform()` 定义为方法。这么一来，关于一张图片的所有数据和可执行的操作都存储在了统一的位置。
- 用最简单的方式解决问题。使用字典、列表和元组往往要比使用模块更加简单、简洁且快速。而使用类则更为复杂。

创始人 Guido 的建议：

> 不要过度构建数据结构。尽量使用元组（以及命名元组）而不是对象。尽量使用简单的属性域而不是 getter/setter 函数……内置数据类型是你最好的朋友。尽可能多地使用数字、字符串、元组、列表、集合以及字典。多看看容器库提供的类型，尤其是双端队列。
>
> —— Guido van Rossum

命名元组

由于 Guido 刚刚提到了*命名元组*（named tuple），那么我们就在这里谈一谈关于它的事情。命名元组是元组的子类，你既可以通过名称（使用 `.name`）来访问其中的值，也可以通过位置进行访问（使用 `[offset]`）。

我们来把前面例子中的 `Duck` 类改写成命名元组，简洁起见，把 `bill` 和 `tail` 当作简单的字符串特性而不当作类。我们可以通过将下面两个参数传入 `namedtuple` 函数来创建命名元组：

- 名称；
- 由多个域名组成的字符串，各个域名之间由空格隔开。

命名元组并不是 Python 自动支持的类型，使用之前需要加载与其相关的模块，下面例子中

的第一行就是在进行模块加载工作：

```
>>> from collections import namedtuple
>>> Duck = namedtuple('Duck', 'bill tail')
>>> duck = Duck('wide orange', 'long')
>>> duck
Duck(bill='wide orange', tail='long')
>>> duck.bill
'wide orange'
>>> duck.tail
'long'
```

也可以用字典来构造一个命名元组：

```
>>> parts = {'bill': 'wide orange', 'tail': 'long'}
>>> duck2 = Duck(**parts)
>>> duck2
Duck(bill='wide orange', tail='long')
```

注意，上面例子中的 `**parts`，它是个关键词变量（keyword argument）。它的作用是将 `parts` 字典中的键和值抽取出来作为参数提供给 `Duck()` 使用。它与下面这行代码的功能一样：

```
>>> duck2 = Duck(bill = 'wide orange', tail = 'long')
```

命名元组是不可变的，但你可以替换其中某些域的值并返回一个新的命名元组：

```
>>> duck3 = duck2._replace(tail='magnificent', bill='crushing')
>>> duck3
Duck(bill='crushing', tail='magnificent')
```

假设我们把 `duck` 定义为字典：

```
>>> duck_dict = {'bill': 'wide orange', 'tail': 'long'}
>>> duck_dict
{'tail': 'long', 'bill': 'wide orange'}
```

可以向字典里添加新的域（键值对）：

```
>>> duck_dict['color'] = 'green'
>>> duck_dict
{'color': 'green', 'tail': 'long', 'bill': 'wide orange'}
```

但无法对命名元组这么做：

```
>>> duck.color = 'green'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'dict' object has no attribute 'color'
```

作为总结，我列出了一些使用命名元组的好处：

- 它无论看起来还是使用起来都和不可变对象非常相似；
- 与使用对象相比，使用命名元组在时间和空间上效率更高；
- 可以使用点号（`.`）对特性进行访问，而不需要使用字典风格的方括号；
- 可以把它作为字典的键。

6.15 练习

(1) 创建一个名为 Thing 的空类并将它打印出来。接着，创建一个属于该类的对象 example，同样将它打印出来。看看这两次打印的结果是一样的还是不同的?

(2) 创建一个新的类 Thing2，将 'abc' 赋值给类特性 letters，打印 letters。

(3) 再创建一个新的类，叫作 Thing3。这次将 'xyz' 赋值给实例（对象）特性 letters，并试着打印 letters。看看你是不是必须先创建一个对象才可以进行打印操作?

(4) 创建一个名为 Element 的类，它包含实例特性 name、symbol 和 number。使用 'Hydrogen'、'H' 和 1 实例化一个对象。

(5) 创建一个字典，包含这些键值对：'name': 'Hydrogen'、'symbol': 'H' 和 'number': 1。然后用这个字典实例化 Element 类的对象 hydrogen。

(6) 为 Element 类定义一个 dump() 方法，用于打印对象的特性（name、symbol 和 number）。使用这个新定义的类创建一个对象 hydrogen 并用 dump() 打印。

(7) 调用 print(hydrogen)，然后修改 Element 的定义，将 dump 方法的名字改为 __str__。再次创建一个 hydrogen 对象并调用 print(hydrogen)，观察输出结果。

(8) 修改 Element 使得 name、symbol 和 number 特性都变成私有的。为它们各定义一个 getter 属性来返回各自的值。

(9) 定义三个类 Bear、Rabbit 和 Octothorpe。对每个类都只定义一个方法 eats()，分别返回 'berries'（Bear）、'clover'（Rabbit）和 'campers'（Octothorpe）。为每个类创建一个对象并输出它们各自吃的食物（调用 eats()）。

(10) 定义三个类 Laser、Claw 以及 SmartPhone。每个类都仅有一个方法 does()，分别返回 'disintegrate'（Laser）、'crush'（Claw）以及 'ring'（SmartPhone）。接着，定义 Robot 类，包含上述三个类的实例（对象）各一个。给 Robot 定义 does() 方法用于输出它各部分的功能。

第 7 章

像高手一样玩转数据

本章将学到许多操作数据的方法，它们大多与下面这两种内置的 Python 数据类型有关。

- 字符串

  Unicode 字符组成的序列，用于存储文本数据。

- 字节和字节数组

  8 比特整数组成的序列，用于存储二进制数据。

7.1 文本字符串

对大多数读者来说，文本应该是最熟悉的数据类型了，因此我们从文本入手，首先介绍一些 Python 中有关字符串操作的强大特性。

7.1.1 Unicode

到目前为止，书中例子使用的都是用原始 ASCII 编码的字符串。ASCII 诞生于 20 世纪 60 年代，那时的计算机还和冰箱差不多大，运算速度也仅仅比人力稍快一些。众所周知，计算机的基本存储单元是字节（byte），它包含 8 位 / 比特（bit），可以存储 256 种不同的值。出于一些设计目的，ASCII 只使用了 7 位（128 种取值）：26 个大写字母、26 个小写字母、10 个阿拉伯数字、一些标点符号、空白符以及一些不可打印的控制符。

不幸的是，世界上现存的字符远远超过了 ASCII 所能支持的 128 个。设想在一个只有 ASCII 字符的世界中，你可以在咖啡厅点个热狗作为晚餐，但永远也点不到美味的 Gewürztraminer 酒[1]。为了支持更多的字母及符号，人们已经做出了许多努力，其中有些成

注 1：这个词在德语中原本含有曲音符，但传到法国时就丢失了。

果你可能见到过。例如下面这两个：

- Latin-1 或 ISO 8859-1
- Windows code page 1252

上面这些编码规则使用全 8 比特（ASCII 只使用了 7 比特）进行编码，但这明显不够用，尤其是当你需要表示非印欧语系的语言符号时。Unicode 编码是一种正在发展中的国际化规范，它可以包含世界上所有语言以及来自数学领域和其他领域的各种符号。

> Unicode 为每个字符赋予了一个特殊的数字编码，这些编码与具体平台、程序、语言均无关。
>
> —— Unicode 协会

Unicode Code Charts 页面（http://www.unicode.org/charts/）包含了通往目前已定义的所有字符集的链接，且包含字符图示。最新的版本（6.2）定义了超过 110 000 种字符，每一种都有自己独特的名字和标识数。这些字符被分成了若干个 8 比特的集合，我们称之为*平面*（plane）。前 256 个平面为*基本多语言平面*（basic multilingual plane）。你可以在维基百科中查看更多关于 Unicode 平面的信息（http://en.wikipedia.org/wiki/Plane_(Unicode)）。

1. Python 3中的Unicode字符串

Python 3 中的字符串是 Unicode 字符串而不是字节数组。这是与 Python 2 相比最大的差别。在 Python 2 中，我们需要区分普通的以字节为单位的字符串以及 Unicode 字符串。

如果你知道某个字符的 Unicode ID，可以直接在 Python 字符串中引用这个 ID 获得对应字符。下面是几个例子。

- 用 `\u` 及 4 个十六进制的数字 [2] 可以从 Unicode 256 个基本多语言平面中指定某一特定字符。其中，前两个十六进制数字用于指定平面号（`00` 到 `FF`），后面两个数字用于指定该字符位于平面中的位置索引。`00` 号平面即为原始的 ASCII 字符集，字符在该平面的位置索引与它的 ASCII 编码一致。
- 我们需要使用更多的比特位来存储那些位于更高平面的字符。Python 为此而设计的转义序列以 `\U` 开头，后面紧跟着 8 个十六进制的数字，其中最左一位需为 `0`。
- 你也可以通过 `\N{name}` 来引用某一字符，其中 `name` 为该字符的标准名称，这对所有平面的字符均适用。在 Unicode 字符名称索引页（http://www.unicode.org/charts/charindex.html）可以查到字符对应的标准名称。

Python 中的 `unicodedata` 模块提供了下面两个方向的转换函数：

- `lookup()`——接受不区分大小写的标准名称，返回一个 Unicode 字符；
- `name()`——接受一个 Unicode 字符，返回大写形式的名称。

下面的例子中，我们将编写一个测试函数，它接受一个 Python Unicode 字符，查找它对应的名称，再用这个名称查找对应的 Unicode 字符（它应该与原始字符相同）：

注 2：0~9、A~F，共 16 个字符。

```
>>> def unicode_test(value):
...     import unicodedata
...     name = unicodedata.name(value)
...     value2 = unicodedata.lookup(name)
...     print('value="%s", name="%s", value2="%s"' % (value, name, value2))
...
```

用一些字符来测试一下吧。首先试一下纯 ASCII 字符：

```
>>> unicode_test('A')
value="A", name="LATIN CAPITAL LETTER A", value2="A"
```

ASCII 标点符号：

```
>>> unicode_test('$')
value="$", name="DOLLAR SIGN", value2="$"
```

Unicode 货币字符：

```
>>> unicode_test('\u00a2')
value="¢", name="CENT SIGN", value2="¢"
```

另一个 Unicode 货币字符：

```
>>> unicode_test('\u20ac')
value="€", name="EURO SIGN", value2="€"
```

这些例子唯一可能遇到的问题来源于使用的字体自身的限制。没有任何一种字体涵盖了所有 Unicode 字符，当缺失对应字符的图片时，会以占位符的形式显示。例如下面是尝试打印 SNOWMAN 字符得到的结果，这里使用的是 dingbat 字体：

```
>>> unicode_test('\u2603')
value="☃", name="SNOWMAN", value2="☃"
```

假设想在 Python 字符串中存储 café 这个词。一种方式是从其他文件或者网站中复制粘贴出来，但这并不一定成功，只能祈祷一切正常：

```
>>> place = 'café'
>>> place
'café'
```

示例中之所以成功是因为我是从以 UTF-8 编码（马上你就会了解）的文本源复制粘贴过来的。

有没有什么办法能够直接指定末尾的 é 字符呢？如果你查看了 E 索引（http://www.unicode.org/charts/charindex.html#E）下的字符会发现，我们所需字符 E WITH ACUTE, LATIN SMALL LETTER 对应的 Unicode 值为 00E9。我们用刚刚的 name() 函数和 lookup() 函数来检测一下，首先用编码值查询字符名称：

```
>>> unicodedata.name('\u00e9')
'LATIN SMALL LETTER E WITH ACUTE'
```

接着，通过名称查询对应的编码值：

```
>>> unicodedata.lookup('E WITH ACUTE, LATIN SMALL LETTER')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: "undefined character name 'E WITH ACUTE, LATIN SMALL LETTER'"
```

为了方便查阅，Unicode 字符名称索引页列出的字符名称是经过修改的，因此与由 `name()` 函数得到的名称有所不同。如果需要将它们转化为真实的 Unicode 名称（Python 使用的），只需将逗号舍去，并将逗号后面的内容移到最前面即可。据此，我们应将 `E WITH ACUTE, LATIN SMALL LETTER` 改为 `LATIN SMALL LETTER E WITH ACUTE`：

```
>>> unicodedata.lookup('LATIN SMALL LETTER E WITH ACUTE')
'é'
```

现在，可以通过字符名称或者编码值来指定 `café` 这个词了：

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> place = 'caf\N{LATIN SMALL LETTER E WITH ACUTE}'
>>> place
'café'
```

上面的代码中，我们将 é 直接插入了字符串中。也可以使用拼接来构造字符串：

```
>>> u_umlaut = '\N{LATIN SMALL LETTER U WITH DIAERESIS}'
>>> u_umlaut
'ü'
>>> drink = 'Gew' + u_umlaut + 'rztraminer'
>>> print('Now I can finally have my', drink, 'in a', place)
Now I can finally have my Gewürztraminer in a café
```

字符串函数 `len` 可以计算字符串中 Unicode 字符的个数，而不是字节数：

```
>>> len('$')
1
>>> len('\U0001f47b')
1
```

2. 使用UTF-8编码和解码

对字符串进行处理时，并不需要在意 Python 中 Unicode 字符的存储细节。

但当需要与外界进行数据交互时则需要完成两件事情：

- 将字符串*编码*为字节；
- 将字节*解码*为字符串。

如果 Unicode 包含的字符种类不超过 64 000 种，我们就可以将字符 ID 统一存储在 2 字节中。遗憾的是，Unicode 所包含的字符种类远不止于此。诚然，我们可以将字符 ID 统一编码在 3 或 4 字节中，但这会使空间开销（内存和硬盘）增加 3 到 4 倍。

两位为 Unix 开发者所熟知的大神 Ken Thompson 和 Rob Pike 在新泽西共用晚餐时解决了

这个问题，他们在餐桌垫上设计出了 UTF-8 动态编码方案。这种方案会动态地为每一个 Unicode 字符分配 1 到 4 字节不等：

- 为 ASCII 字符分配 1 字节；
- 为拉丁语系（除西里尔语）的语言分配 2 字节；
- 为其他的位于基本多语言平面的字符分配 3 字节；
- 为剩下的字符集分配 4 字节，这包括一些亚洲语言及符号。

UTF-8 是 Python、Linux 以及 HTML 的标准文本编码格式。这种编码方式简单快速、字符覆盖面广、出错率低。在代码中全都使用 UTF-8 编码会是一种非常棒的体验，你再也不需要不停地转化各种编码格式。

> 如果你创建 Python 字符串时使用了从别的文本源（例如网页）复制粘贴过来的字符串，一定要确保文本源使用的是 UTF-8 编码。将 Latin-1 或者 Windows 1252 复制粘贴为 Python 字符串的错误极其常见，这样得到的字节序列是无效的，会产生许多后续隐患。

3. 编码

编码是将字符串转化为一系列字节的过程。字符串的 `encode()` 函数所接收的第一个参数是编码方式名。可选的编码方式列在了表 7-1 中。

表7-1：编码方式

编码	说明
`'ascii'`	经典的 7 比特 ASCII 编码
`'utf-8'`	最常用的以 8 比特为单位的变长编码
`'latin-1'`	也被称为 ISO 8859-1 编码
`'cp-1252'`	Windows 常用编码
`'unicode-escape'`	Python 中 Unicode 的转义文本格式，`\uxxxx` 或者 `\Uxxxxxxxx`

你可以将任何 Unicode 数据以 UTF-8 的方式进行编码。我们试着将 Unicode 字符串 `'\u2603'` 赋值给 `snowman`：

```
>>> snowman = '\u2603'
```

`snowman` 是一个仅包含一个字符的 Unicode 字符串，这与它存储所需的字节数没有任何关系：

```
>>> len(snowman)
1
```

下一步将这个 Unicode 字符编码为字节序列：

```
>>> ds = snowman.encode('utf-8')
```

就像我之前提到的，UTF-8 是一种变长编码方式。在这个例子中，单个 Unicode 字符 `snowman` 占用了 3 字节的空间：

```
>>> len(ds)
3
```

```
>>> ds
b'\xe2\x98\x83'
```

现在，len() 返回了字节数（3），因为 ds 是一个 bytes 类型的变量。

当然，你也可以使用 UTF-8 以外的编码方式，但该 Unicode 字符串有可能无法被指定的编码方式处理，此时 Python 会抛出异常。例如，如果你想要使用 ascii 方式进行编码，必须保证待编码的字符串仅包含 ASCII 字符集里的字符，不含有任何其他的 Unicode 字符，否则会出现错误：

```
>>> ds = snowman.encode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeEncodeError: 'ascii' codec can't encode character '\u2603'
in position 0: ordinal not in range(128)
```

encode() 函数可以接受额外的第二个参数来帮助你避免编码异常。它的默认值是 'strict'，如上例所示，当函数检测到需要处理的字符串包含非 ASCII 字符时，会抛出 UnicodeEncodeError 异常。当然，该参数还有别的可选值，例如 'ignore' 会抛弃任何无法进行编码的字符：

```
>>> snowman.encode('ascii', 'ignore')
b''
```

'replace' 会将所有无法进行编码的字符替换为 ?：

```
>>> snowman.encode('ascii', 'replace')
b'?'
```

'backslashreplace' 则会创建一个和 unicode-escape 类似的 Unicode 字符串：

```
>>> snowman.encode('ascii', 'backslashreplace')
b'\\u2603'
```

如果你需要一份 Unicode 转义符序列的可打印版本，可以考虑使用上面这种方式。

下面的代码可以用于创建网页中使用的字符实体串：

```
>>> snowman.encode('ascii', 'xmlcharrefreplace')
b'&#9731;'
```

4. 解码

*解码*是将字节序列转化为 Unicode 字符串的过程。我们从外界文本源（文件、数据库、网站、网络 API 等）获得的所有文本都是经过编码的字节串。重要的是需要知道它是以何种方式编码的，这样才能逆转编码过程以获得 Unicode 字符串。

问题是字节串本身不带有任何指明编码方式的信息。之前我也提到过从网站随意复制粘贴文本的风险，你也可能遇到过网页乱码的情况，本应是 ASCII 字符的位置却被奇怪的字符占据了，这些都是编码和解码的方式不一致导致的。

创建一个 place 字符串，赋值为 'café'：

```
>>> place = 'caf\u00e9'
>>> place
'café'
>>> type(place)
<class 'str'>
```

将它以 UTF-8 格式编码为 `bytes` 型变量，命名为 `place_bytes`：

```
>>> place_bytes = place.encode('utf-8')
>>> place_bytes
b'caf\xc3\xa9'
>>> type(place_bytes)
<class 'bytes'>
```

注意，`place_bytes` 包含 5 个字节。前 3 个字节的内容与 ASCII 一样（UTF-8 的强大之处），最后两个字节用于编码 `'é'`。现在，将字节串转换回 Unicode 字符串：

```
>>> place2 = place_bytes.decode('utf-8')
>>> place2
'café'
```

一切正常，这是因为编码和解码使用的都是 UTF-8 格式。如果使用其他格式进行解码会发生什么？

```
>>> place3 = place_bytes.decode('ascii')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
UnicodeDecodeError: 'ascii' codec can't decode byte 0xc3 in position 3:
ordinal not in range(128)
```

ASCII 解码器会抛出异常，因为字节值 `0xc3` 在 ASCII 编码中是非法值。对于另一些使用 8 比特编码的方式而言，位于 128（十六进制 `80`）到 255（十六进制 `FF`）之间的 8 比特的字符集可能是合法的，但解码得到的结果显然与 UTF-8 不同：

```
>>> place4 = place_bytes.decode('latin-1')
>>> place4
'cafÃ©'
>>> place5 = place_bytes.decode('windows-1252')
>>> place5
'cafÃ©'
```

这个故事告诉我们：尽可能统一使用 UTF-8 编码。况且它出错率低，兼容性好，可以表达所有的 Unicode 字符，编码和解码的速度又快，这么多优点，何乐而不为？

5. 更多内容

如果想要了解更多关于 Unicode 的细节，下面这些链接对你可能会有所帮助：

- Unicode HOWTO（https://docs.python.org/3/howto/unicode.html）
- Pragmatic Unicode（http://nedbatchelder.com/text/unipain.html）
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)（http://www.joelonsoftware.com/articles/Unicode.html）

7.1.2 格式化

之前几乎都没有提到过文本格式化的问题，现在有必要关注一下这方面的内容了。第 2 章曾经用过一些字符串排版函数，那些示例代码要么简单地使用 `print()` 语句，要么直接在交互式解释器显示。现在是时候看看如何使用不同的格式化方法将变量*插值*（interpolate）到字符串中了，即将变量的值嵌入字符串中。你可以用这种方法来生成那些格式框架看起来一样的报告或者其他固定格式的输出。

Python 有两种格式化字符串的方式，我们习惯简单地称之为*旧式*（old style）和*新式*（new style）。这两种方式在 Python 2 和 Python 3 中都适用（新式格式化方法适用于 Python 2.6 及以上）。旧式格式化相对简单些，因此我们从它开始。

1. 使用%的旧式格式化

旧式格式化的形式为 `string % data`。其中 `string` 包含的是待插值的序列。表 7-2 展示了最简单的插值序列，它仅由 `%` 以及一个用于指定数据类型的字母组成。

表7-2：转换类型

`%s`	字符串
`%d`	十进制整数
`%x`	十六进制整数
`%o`	八进制整数
`%f`	十进制浮点数
`%e`	以科学计数法表示的浮点数
`%g`	十进制或科学计数法表示的浮点数
`%%`	文本值 `%` 本身

下面是一些简单的例子。首先格式化一个整数：

```
>>> '%s' % 42
'42'
>>> '%d' % 42
'42'
>>> '%x' % 42
'2a'
>>> '%o' % 42
'52'
```

接着是浮点数：

```
>>> '%s' % 7.03
'7.03'
>>> '%f' % 7.03
'7.030000'
>>> '%e' % 7.03
'7.030000e+00'
>>> '%g' % 7.03
'7.03'
```

整数和字面值 `%`：

```
>>> '%d%%' % 100
'100%'
```

下面是一些关于字符串和整数的插值操作：

```
>>> actor = 'Richard Gere'
>>> cat = 'Chester'
>>> weight = 28

>>> "My wife's favorite actor is %s" % actor
"My wife's favorite actor is Richard Gere"

>>> "Our cat %s weighs %s pounds" % (cat, weight)
'Our cat Chester weighs 28 pounds'
```

字符串内的 `%s` 意味着需要插入一个字符串。字符串中出现 `%` 的次数需要与 `%` 之后所提供的数据项个数相同。如果只需插入一个数据，例如前面的 `actor`，直接将需要插入的数据置于 `%` 后即可。如果需要插入多个数据，则需要将它们封装进一个元组（以圆括号为界，逗号分开），例如上例中的 `(cat, weight)`。

尽管 `weight` 是一个整数，格式化串中的 `%s` 也会将它转化为字符串型。

你可以在 `%` 和指定类型的字母之间设定最大和最小宽度、排版以及填充字符，等等。

我们来定义一个整数 `n`、一个浮点数 `f` 以及一个字符串 `s`：

```
>>> n = 42
>>> f = 7.03
>>> s = 'string cheese'
```

使用默认宽度格式化它们：

```
>>> '%d %f %s' % (n, f, s)
'42 7.030000 string cheese'
```

为每个变量设定最小域宽为 10 个字符，右对齐，左侧不够用空格填充：

```
>>> '%10d %10f %10s' % (n, f, s)
'        42   7.030000 string cheese'
```

和上面的例子使用同样的域宽，但改成左对齐：

```
>>> '%-10d %-10f %-10s' % (n, f, s)
'42         7.030000   string cheese'
```

这次仍然使用之前的域宽，但是设定最大字符宽度为 4，右对齐。这样的设置会截断超过长度限制的字符串，并且将浮点数的精度限制在小数点后 4 位：

```
>>> '%10.4d %10.4f %10.4s' % (n, f, s)
'      0042     7.0300       stri'
```

去掉最小域宽为 10 的限制：

```
>>> '%.4d %.4f %.4s' % (n, f, s)
'0042 7.0300 stri'
```

最后，改变一下上面例子的硬编码方式，将域宽、字符宽度等设定作为参数：

```
>>> '%*.*d %*.*f %*.*s' % (10, 4, n, 10, 4, f, 10, 4, s)
'      0042     7.0300       stri'
```

2. 使用{}和format的新式格式化

旧式格式化方式现在仍然兼容。Python 2（将永远停止在 2.7 版本）会永远提供对旧式格式化的支持。然而，如果你在使用 Python 3，新式格式化更值得推荐。

新式格式化最简单的用法如下所示：

```
>>> '{} {} {}'.format(n, f, s)
'42 7.03 string cheese'
```

旧式格式化中传入参数的顺序需要与 % 占位符出现的顺序完全一致，但在新式格式化里，可以自己指定插入的顺序：

```
>>> '{2} {0} {1}'.format(f, s, n)
'42 7.03 string cheese'
```

0 代表第一个参数 f；1 代表字符串 s；2 代表最后一个参数，整数 n。

参数可以是字典或者命名变量，格式串中的标识符可以引用这些名称：

```
>>> '{n} {f} {s}'.format(n=42, f=7.03, s='string cheese')
'42 7.03 string cheese'
```

下面的例子中，我们试着将之前作为参数的 3 个值存到一个字典中，如下所示：

```
>>> d = {'n': 42, 'f': 7.03, 's': 'string cheese'}
```

下面的例子中，{0} 代表整个字典，{1} 则代表字典后面的字符串 'other'：

```
>>> '{0[n]} {0[f]} {0[s]} {1}'.format(d, 'other')
'42 7.03 string cheese other'
```

上面这些例子都是以默认格式打印结果的。旧式格式化允许在 % 后指定参数格式，但在新式格式化里，将这些格式标识符放在 : 后。首先使用位置参数的例子：

```
>>> '{0:d} {1:f} {2:s}'.format(n, f, s)
'42 7.030000 string cheese'
```

接着使用相同的值，但这次它们作为命名参数：

```
>>> '{n:d} {f:f} {s:s}'.format(n=42, f=7.03, s='string cheese')
'42 7.030000 string cheese'
```

新式格式化也支持其他各类设置（最小域宽、最大字符宽、排版，等等）。

下面是一个最小域宽设为 10、右对齐（默认）的例子：

```
>>> '{0:10d} {1:10f} {2:10s}'.format(n, f, s)
'        42   7.030000 string cheese'
```

与上面例子一样，但使用 > 字符设定右对齐显然要更为直观：

```
>>> '{0:>10d} {1:>10f} {2:>10s}'.format(n, f, s)
'        42   7.030000 string cheese'
```

最小域宽为 10，左对齐：

```
>>> '{0:<10d} {1:<10f} {2:<10s}'.format(n, f, s)
'42         7.030000   string cheese'
```

最小域宽为 10，居中：

```
>>> '{0:^10d} {1:^10f} {2:^10s}'.format(n, f, s)
'    42      7.030000  string cheese'
```

新式格式化与旧式格式化相比有一处明显的不同：*精度*（precision，小数点后面的数字）对于浮点数而言仍然代表着小数点后的数字个数，对于字符串而言则代表着最大字符个数，但在新式格式化中你无法对整数设定精度：

```
>>> '{0:>10.4d} {1:>10.4f} {2:10.4s}'.format(n, f, s)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: Precision not allowed in integer format specifier
>>> '{0:>10d} {1:>10.4f} {2:>10.4s}'.format(n, f, s)
'        42     7.0300       stri'
```

最后一个可设定的值是*填充字符*。如果想要使用空格以外的字符进行填充，只需把它放在 : 之后，其余任何排版符（<、>、^）和宽度标识符之前即可：

```
>>> '{0:!^20s}'.format('BIG SALE')
'!!!!!!BIG SALE!!!!!!'
```

7.1.3　使用正则表达式匹配

第 2 章接触到一些简单的字符串操作。有了这些知识，你可能已经会在命令行里使用一些简单的“通配符”模式了，例如 ls*.py，这条命令的意思是“列出当前目录下所有以 .py 结尾的文件名”。

是时候使用*正则表达式*（regular expression）探索一些复杂模式匹配的方法了。与之相关的功能都位于标准库模块 `re` 中，因此首先需要引用它。你需要定义一个用于匹配的*模式*（pattern）字符串以及一个匹配的对象：*源*（source）字符串。简单的匹配，如下所示：

```
result = re.match('You', 'Young Frankenstein')
```

这里，`'You'` 是*模式*，`'Young Frankenstein'` 是*源*——你想要检查的字符串。`match()` 函数用于查看源是否以模式开头。

对于更加复杂的匹配，可以先对模式进行编译以加快匹配速度：

```
youpattern = re.compile('You')
```

然后就可以直接使用编译好的模式进行匹配了：

```
result = youpattern.match('Young Frankenstein')
```

`match()` 并不是比较 source 和 pattern 的唯一方法。下面列出了另外一些可用的方法：

- search() 会返回第一次成功匹配，如果存在的话；
- findall() 会返回所有不重叠的匹配，如果存在的话；
- split() 会根据 pattern 将 source 切分成若干段，返回由这些片段组成的列表；
- sub() 还需一个额外的参数 replacement，它会把 source 中所有匹配的 pattern 改成 replacement。

1. 使用match()进行准确匹配

字符串 'Young Frankenstein' 是以单词 'You' 开头的吗？以下是一些带注释的代码：

```
>>> import re
>>> source = 'Young Frankenstein'
>>> m = re.match('You', source)  # 从源字符串的开头开始匹配
>>> if m:  # 匹配成功返回了对象,将它输出看看匹配得到的是什么
...     print(m.group())
...
You
>>> m = re.match('^You', source) # 起始锚点也能起到同样作用
>>> if m:
...     print(m.group())
...
You
```

尝试匹配 'Frank' 又会如何？

```
>>> m = re.match('Frank', source)
>>> if m:
...     print(m.group())
...
```

这一次，match() 什么也没有返回，if 也没有执行内部的 print 语句。如前所述，match() 只能检测以模式串作为开头的源字符串。但是 search() 可以检测任何位置的匹配：

```
>>> m = re.search('Frank', source)
>>> if m:
...      print(m.group())
...
Frank
```

改变一下匹配的模式：

```
>>> m = re.match('.*Frank', source)
>>> if m:  # match返回对象
...     print(m.group())
...
Young Frank
```

以下是对新模式能够匹配成功的简单解释：

- . 代表任何单一字符；
- * 代表任意一个它之前的字符，.* 代表任意多个字符（包括 0 个）；
- Frank 是我们想要在源字符串中某处进行匹配的短语。

match() 返回了匹配 .*Frank 的字符串：'Young Frank'。

2. 使用`search()`寻找首次匹配

你可以使用 `search()` 在源字符串 `'Young Frankenstein'` 的任意位置寻找模式 `'Frank'`，无需通配符 `.*`：

```
>>> m = re.search('Frank', source)
>>> if m:  # search返回对象
...     print(m.group())
...
Frank
```

3. 使用`findall()`寻找所有匹配

之前的例子都是查找到一个匹配即停止。但如果想要知道一个字符串中出现了多少次字母 `'n'` 应该怎么办？

```
>>> m = re.findall('n', source)
>>> m   # findall返回了一个列表
['n', 'n', 'n', 'n']
>>> print('Found', len(m), 'matches')
Found 4 matches
```

将模式改成 `'n'`，紧跟着任意一个字符，结果又如何？

```
>>> m = re.findall('n.', source)
>>> m
['ng', 'nk', 'ns']
```

注意，上面例子中最后一个 `'n'` 并没有匹配成功，需要通过 `?` 说明 `'n'` 后面的字符是可选的：

```
>>> m = re.findall('n.?', source)
>>> m
['ng', 'nk', 'ns', 'n']
```

4. 使用`split()`按匹配切分

下面的示例展示了如何依据模式而不是简单的字符串（就像普通的 `split()` 方法做的）将一个字符串切分成由一系列子串组成的列表：

```
>>> m = re.split('n', source)
>>> m    # split返回的列表
['You', 'g Fra', 'ke', 'stei', '']
```

5. 使用`sub()`替换匹配

这和字符串 `replace()` 方法有些类似，只不过使用的是模式而不是文本串：

```
>>> m = re.sub('n', '?', source)
>>> m   # sub返回的字符串
'You?g Fra?ke?stei?'
```

6. 模式：特殊的字符

许多书中关于正则表达式的描述都是从如何定义它开始的，我觉得这不太符合学习的逻辑。正则表达式不是一两句就能说清楚的小语言，它拥有大量的语言细节，会完全占据你的大脑让你无所适从。它使用的符号实在是太多了，看起来简直就像是幽灵画符一样！

有了上面介绍的方法（match()、search()、findall() 和 sub()）做铺垫，现在可以从应用讲起并研究如何构造正则表达式了，即上述方法中的模式。

已经见过的一些基本模式：

- 普通的文本值代表自身，用于匹配非特殊字符；
- 使用 . 代表任意除 \n 外的字符；
- 使用 * 表示任意多个字符（包括 0 个）；
- 使用 ? 表示可选字符（0 个或 1 个）。

接下来要介绍一些特殊字符，参见表 7-3。

表7-3：特殊字符

模式	匹配
\d	一个数字字符
\D	一个非数字字符
\w	一个字母或数字字符
\W	一个非字母非数字字符
\s	空白符
\S	非空白符
\b	单词边界（一个 \w 与 \W 之间的范围，顺序可逆）
\B	非单词边界

Python 的 string 模块中预先定义了一些可供我们测试用的字符串常量。我们将使用其中的 printable 字符串，它包含 100 个可打印的 ASCII 字符，包括大小写字母、数字、空格符以及标点符号：

```
>>> import string
>>> printable = string.printable
>>> len(printable)
100
>>> printable[0:50]
'0123456789abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMN'
>>> printable[50:]
'OPQRSTUVWXYZ!"#$%&\'()*+,-./:;<=>?@[\\]^_`{|}~ \t\n\r\x0b\x0c'
```

printable 中哪些字符是数字？

```
>>> re.findall('\d', printable)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9']
```

哪些字符是数字、字符或下划线？

```
>>> re.findall('\w', printable)
['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b',
'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n',
'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z',
'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L',
'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X',
'Y', 'Z', '_']
```

哪些属于空格符？

```
>>> re.findall('\s', printable)
[' ', '\t', '\n', '\r', '\x0b', '\x0c']
```

正则表达式不仅仅适用于 ASCII 字符，例如 \d 还可以匹配 Unicode 的数字字符，并不局限于 ASCII 中的 '0' 到 '9'。我们从 FileFormat.info（http://www.fileformat.info/info/unicode/category/Ll/list.htm）中引入两个新的非 ASCII 编码的小写字母。

这个测试例子中，在模式中添加以下内容：

- 三个 ASCII 字母
- 三个不会被 \w 所匹配的标点符号
- Unicode 中的 LATIN SMALL LETTER E WITH CIRCUMFLEX（\u00ea）
- Unicode 中的 LATIN SMALL LETTER E WITH BREVE（\u0115）

```
>>> x = 'abc' + '-/*' + '\u00ea' + '\u0115'
```

与预期的一样，应用这个模式可以匹配出下面这些字母：

```
>>> re.findall('\w', x)
['a', 'b', 'c', 'ê', 'ĕ']
```

7. 模式：使用标识符

现在试着用表 7-4 中所包含的一些常用的模式标识符来烹饪一道“符号比萨”大餐。

表中，expr 和其他斜体的单词表示合法的正则表达式。

表7-4：模式标识符

模式	匹配
abc	文本值 abc
(*expr*)	*expr*
*expr*1 | *expr*2	*expr*1 或 *expr*2
.	除 \n 外的任何字符
^	源字符串的开头
$	源字符串的结尾
prev?	0 个或 1 个 *prev*
*prev**	0 个或多个 *prev*，尽可能多地匹配
*prev**?	0 个或多个 *prev*，尽可能少地匹配
prev+	1 个或多个 *prev*，尽可能多地匹配
prev+?	1 个或多个 *prev*，尽可能少地匹配
prev{m}	*m* 个连续的 *prev*
prev{m, n}	*m* 到 *n* 个连续的 *prev*，尽可能多地匹配
prev{m, n}?	*m* 到 *n* 个连续的 *prev*，尽可能少地匹配
[abc]	a 或 b 或 c（和 a|b|c 一样）
[^abc]	非（a 或 b 或 c）

（续）

模式	匹配
prev (?=*next*)	如果后面为 *next*，返回 *prev*
prev (?!*next*)	如果后面非 *next*，返回 *prev*
(?<=*prev*) *next*	如果前面为 *prev*，返回 *next*
(?<!*prev*) *next*	如果前面非 *prev*，返回 *next*

在看下面的例子时，你可能需要时不时地查阅上面的表格。先来定义我们使用的源字符串：

```
>>> source = '''I wish I may, I wish I might
... Have a dish of fish tonight.'''
```

首先，在源字符串中检索 `wish`：

```
>>> re.findall('wish', source)
['wish', 'wish']
```

接着，对源字符串任意位置查询 `wish` 或者 `fish`：

```
>>> re.findall('wish|fish', source)
['wish', 'wish', 'fish']
```

从字符串开头开始匹配 `wish`：

```
>>> re.findall('^wish', source)
[]
```

从字符串开头开始匹配 `I wish`：

```
>>> re.findall('^I wish', source)
['I wish']
```

从字符串结尾开始匹配 `fish`：

```
>>> re.findall('fish$', source)
[]
```

最后，从字符串结尾开始匹配 `fish tonight.`：

```
>>> re.findall('fish tonight.$', source)
['fish tonight.']
```

`^` 和 `$` 叫作锚点（anchor）：`^` 将搜索域定位到源字符串的开头，`$` 则定位到末尾。上面例子中的 `.$` 可以匹配末尾的任意字符，包括句号，因此能成功匹配。但更准确地说，上面的例子应该使用转义符将 `.` 转义为句号，这才是我们真正想示意的纯文本值匹配：

```
>>> re.findall('fish tonight\.$', source)
['fish tonight.']
```

接下来查询以 `w` 或 `f` 开头，后面紧接着 `ish` 的匹配：

```
>>> re.findall('[wf]ish', source)
['wish', 'wish', 'fish']
```

查询以若干个 w、s 或 h 组合的匹配：

```
>>> re.findall('[wsh]+', source)
['w', 'sh', 'w', 'sh', 'h', 'sh', 'sh', 'h']
```

查询以 ght 开头，后面紧跟一个非数字非字母字符的匹配：

```
>>> re.findall('ght\W', source)
['ght\n', 'ght.']
```

查询以 I 开头，后面跟着 wish 的匹配（wish 出现次数尽量少）：

```
>>> re.findall('I (?=wish)', source)
['I ', 'I ']
```

最后查询以 wish 结尾，前面为 I 的匹配（I 出现的次数尽量少）：

```
>>> re.findall('(?<=I) wish', source)
[' wish', ' wish']
```

有时，正则表达式的语法可能会与 Python 本身的语法冲突。例如，我们期望下面例子中的模式能匹配任何以 fish 开头的词：

```
>>> re.findall('\bfish', source)
[]
```

为什么没有匹配成功？第 2 章曾提到，Python 字符串会使用一些特殊的*转义符*。例如上面的 \b，它在字符串中代表退格，但在正则表达式中，它代表一个单词的开头位置。因此，把 Python 的*普通字符串*用作正则表达式的模式串时需要特别注意，不要像上面一样与转义符产生冲突。或者在任何使用正则表达式的地方都记着在模式串的前面添加字符 r，这样可以告诉 Python 这是一个正则表达式，从而禁用字符串转义符，如下所示：

```
>>> re.findall(r'\bfish', source)
['fish']
```

8. 模式：定义匹配的输出

当使用 match() 或 search() 时，所有的匹配会以 m.group() 的形式返回到对象 m 中。如果你用括号将某一模式包裹起来，括号中模式匹配得到的结果归入自己的 group（无名称）中，而调用 m.groups() 可以得到包含这些匹配的元组，如下所示：

```
>>> m = re.search(r'(. dish\b).*(\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
```

(?P< name >expr) 这样的模式会匹配 expr，并将匹配结果存储到名为 name 的组中：

```
>>> m = re.search(r'(?P<DISH>. dish\b).*(?P<FISH>\bfish)', source)
>>> m.group()
'a dish of fish'
>>> m.groups()
('a dish', 'fish')
>>> m.group('DISH')
```

```
'a dish'
>>> m.group('FISH')
'fish'
```

7.2　二进制数据

处理文本数据比较晦涩难懂（新旧格式化、正则表达式等），而处理二进制数据就有趣多了。你需要了解像字节序（endianness，电脑处理器是如何将数据组织存储为字节的）以及整数的符号位（sign bit）之类的概念。你可能需要研究二进制文件格式、网络包等内容，从而对其中的数据进行提取甚至修改。本节将了解到 Python 中有关二进制数据的一些基本操作。

7.2.1　字节和字节数组

Python 3 引入了下面两种使用 8 比特序列存储小整数的方式，每 8 比特可以存储从 0~255 的值：

- 字节是不可变的，像字节数据组成的元组；
- 字节数组是可变的，像字节数据组成的列表。

我们的示例从创建列表 `blist` 开始。接着需使用这个列表创建一个 `bytes` 类型的变量 `the_bytes` 以及一个 `bytearray` 类型的变量 `the_byte_array`：

```
>> blist = [1, 2, 3, 255]
>>> the_bytes = bytes(blist)
>>> the_bytes
b'\x01\x02\x03\xff'
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
```

`bytes` 类型值的表示形式比较特殊：以 `b` 开头，接着是一个单引号，后面跟着由十六进制数（例如 `\x02`）或 ASCII 码组成的序列，最后以配对的单引号结束。Python 会将这些十六进制数或者 ASCII 码转换为整数，如果该字节的值为有效 ASCII 编码则会显示 ASCII 字符。

```
>>> b'\x61'
 b'a'

>>> b'\x01abc\xff'
 b'\x01abc\xff'
```

下面的例子说明了 `bytes` 类型的不可变性：

```
>>> the_bytes[1] = 127
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'bytes' object does not support item assignment
```

但 `bytearray` 类型的变量是可变的：

```
>>> the_byte_array = bytearray(blist)
>>> the_byte_array
bytearray(b'\x01\x02\x03\xff')
>>> the_byte_array[1] = 127
>>> the_byte_array
bytearray(b'\x01\x7f\x03\xff')
```

下面两行代码都会创建一个包含 256 个元素的结果，包含 0~255 的所有值：

```
>>> the_bytes = bytes(range(0, 256))
>>> the_byte_array = bytearray(range(0, 256))
```

打印 `bytes` 或 `bytearray` 数据时，Python 会以 `\x`*xx* 的形式表示不可打印的字符，以 ASCII 字符的形式表示可打印的字符（以及一些转义字符，例如 `\n` 而不是 `\x0a`）。下面是 `the_bytes` 的打印结果（手动设置为一行显示 16 个字节）：

```
>>> the_bytes
b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\t\n\x0b\x0c\r\x0e\x0f
\x10\x11\x12\x13\x14\x15\x16\x17\x18\x19\x1a\x1b\x1c\x1d\x1e\x1f
!"#$%&\'()*+,-./
0123456789:;<=>?
@ABCDEFGHIJKLMNO
PQRSTUVWXYZ[\\]^_
`abcdefghijklmno
pqrstuvwxyz{|}~\x7f
\x80\x81\x82\x83\x84\x85\x86\x87\x88\x89\x8a\x8b\x8c\x8d\x8e\x8f
\x90\x91\x92\x93\x94\x95\x96\x97\x98\x99\x9a\x9b\x9c\x9d\x9e\x9f
\xa0\xa1\xa2\xa3\xa4\xa5\xa6\xa7\xa8\xa9\xaa\xab\xac\xad\xae\xaf
\xb0\xb1\xb2\xb3\xb4\xb5\xb6\xb7\xb8\xb9\xba\xbb\xbc\xbd\xbe\xbf
\xc0\xc1\xc2\xc3\xc4\xc5\xc6\xc7\xc8\xc9\xca\xcb\xcc\xcd\xce\xcf
\xd0\xd1\xd2\xd3\xd4\xd5\xd6\xd7\xd8\xd9\xda\xdb\xdc\xdd\xde\xdf
\xe0\xe1\xe2\xe3\xe4\xe5\xe6\xe7\xe8\xe9\xea\xeb\xec\xed\xee\xef
\xf0\xf1\xf2\xf3\xf4\xf5\xf6\xf7\xf8\xf9\xfa\xfb\xfc\xfd\xfe\xff'
```

看起来可能有些困惑，毕竟上面输出的数据是字节（小整数）而不是字符。

7.2.2　使用struct转换二进制数据

如你所见，Python 中有许多文本处理工具（模块、函数等），然而处理二进制数据的工具则要少得多。标准库里有一个 `struct` 模块，专门用于处理类似 C 和 C++ 中结构体的数据。你可以使用 `struct` 模块的功能将二进制数据转换为 Python 中的数据结构。

以一个 PNG 文件（一种常见的图片格式，其他图片格式还有 GIF、JPEG 等）为例看看 `struct` 是如何工作的。我们来编写一个小程序，从 PNG 文件中获得图片的宽度和高度信息。

使用 O'Reilly 的经典标志：一只睁大了眼睛的眼镜猴，见图 7-1。

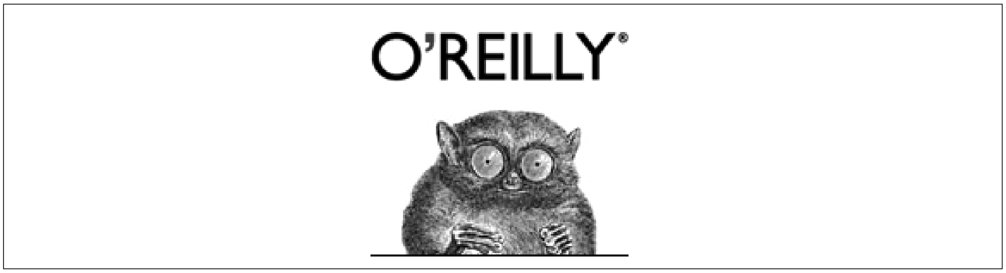


图 7-1：O'Reilly 的标志眼镜猴

你可以在 Wikipedia（http://upload.wikimedia.org/wikipedia/en/9/95/O'Reilly_logo.png）上获取这张图片的 PNG 文件。第 8 章之前都不会讨论读取文件的方法，因此这里我仅仅是将这个文件下载下来，并编写了一个简单的小程序将它的数据以字节形式打印出来，然后将起始的 30 字节数据存入 Pythonbytes 型变量 data 中，如下所示。方便起见，你只需复制这部分数据即可。（PNG 格式规定了图片的宽度和高度信息存储在初始 24 字节中，因此不需要其他的额外数据。）

```
>>> import struct
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> if data[:8] == valid_png_header:
...     width, height = struct.unpack('>LL', data[16:24])
...     print('Valid PNG, width', width, 'height', height)
... else:
...     print('Not a valid PNG')
...
Valid PNG, width 154 height 141
```

以上代码说明：

- data 包含了 PNG 文件的前 30 字节内容，为了书的排版，我将这 30 字节数据放到了两行字节串中，并用 + 和续行符（\）将它们连接起来；
- valid_png_header 包含 8 字节序列，它标志着这是一个有效的 PNG 格式文件；
- width 值位于第 16~20 字节，height 值则位于第 21~24 字节。

上面代码中的 >LL 是一个格式串，它用于指导 unpack() 正确解读字节序列并将它们组装成 Python 中的数据类型。可以将它分解成下面几个基本格式标志：

- > 用于指明整数是以大端（big-endian）方案存储的；
- 每个 L 代表一个 4 字节的无符号长（unsigned long）整数。

你可以直接获取 4 字节数据：

```
>>> data[16:20]
b'\x00\x00\x00\x9a'
>>> data[20:24]0x9a
b'\x00\x00\x00\x8d'
```

大端方案将高字节放在左侧。由于宽度和高度都小于 255，因此它们存储在每一个 4 字节序列的最后一字节中。不难验证，上面的十六进制数转换为十进制后与我们预期的数值（图片的宽和高）一致：

```
>>> 0x9a
154
>>> 0x8d
141
```

如果想要执行上述过程的逆过程，将 Python 数据转换为字节，可以使用 `struct pack()` 函数：

```
>>> import struct
>>> struct.pack('>L', 154)
b'\x00\x00\x00\x9a'
>>> struct.pack('>L', 141)
b'\x00\x00\x00\x8d'
```

表 7-5 和表 7-6 列出了 `pack()` 和 `unpack()` 使用的一些格式标识符。

首先是字节序标识符。

表7-5：字节序标识符

标识符	字节序
<	小端方案
>	大端方案

表7-6：格式标识符

标识符	描述	字节
x	跳过一个字节	1
b	有符号字节	1
B	无符号字节	1
h	有符号短整数	2
H	无符号短整数	2
i	有符号整数	4
I	无符号整数	4
l	有符号长整数	4
L	无符号长整数	4
Q	无符号 long long 型整数	8
f	单精度浮点数	4
d	双精度浮点数	8
p	数量和字符	1 + 数量
s	字符	数量

类型标识符紧跟在字节序标识符的后面。任何标识符的前面都可以添加数字用于指定需要匹配的数量，例如 `5B` 代表 `BBBBB`。

可以使用数量前缀改写 >LL：

```
>>> struct.unpack('>2L', data[16:24])
(154, 141)
```

之前的例子中使用了切片 data[16:24] 直接获取所需的特定字节，也可以使用 x 标识符来跳过不需要的字节：

```
>>> struct.unpack('>16x2L6x', data)
(154, 141)
```

上面格式串的含义如下：

- 使用大端方案（>）
- 跳过 16 个字节（16x）
- 读取 8 字节内容——两个无符号长整数（2L）
- 跳过最后 6 个字节（6x）

7.2.3 其他二进制数据工具

一些第三方开源包提供了下面这些更加直观地定义和提取二进制数据的方法：

- bitstring（https://code.google.com/p/python-bitstring/）
- construct（http://construct.readthedocs.org/en/latest/）
- hachoir（https://bitbucket.org/haypo/hachoir/wiki/Home）
- binio（http://spika.net/py/binio/）

你可以在附录 D 查看下载和安装外部包的详细过程，这里不再赘述。接下来的几个例子需要提前安装 construct 包，只需执行下面这行代码即可：

```
$ pip install construct
```

下面的例子展示了如何使用 construct 从之前的 data 中提取 PNG 图片的尺寸：

```
>>> from construct import Struct, Magic, UBInt32, Const, String
>>> # 基于https://github.com/construct上的代码修改而来
>>> fmt = Struct('png',
...     Magic(b'\x89PNG\r\n\x1a\n'),
...     UBInt32('length'),
...     Const(String('type', 4), b'IHDR'),
...     UBInt32('width'),
...     UBInt32('height')
...     )
>>> data = b'\x89PNG\r\n\x1a\n\x00\x00\x00\rIHDR' + \
...     b'\x00\x00\x00\x9a\x00\x00\x00\x8d\x08\x02\x00\x00\x00\xc0'
>>> result = fmt.parse(data)
>>> print(result)
Container:
    length = 13
    type = b'IHDR'
    width = 154
    height = 141
```

```
>>> print(result.width, result.height)
154, 141
```

7.2.4　使用binascii()转换字节/字符串

标准 binascii 模块提供了在二进制数据和多种字符串表示（十六进制、六十四进制、uuencoded，等等）之间转换的函数。例如，下面的小例子将 8- 字节的 PNG 头打印为十六进制值的形式，而不是 Python 默认的打印 bytes 型变量的方式：混合使用 ASCII 和转义的 \x *xx*。

```
>>> import binascii
>>> valid_png_header = b'\x89PNG\r\n\x1a\n'
>>> print(binascii.hexlify(valid_png_header))
b'89504e470d0a1a0a'
```

反过来转换也可以：

```
>>> print(binascii.unhexlify(b'89504e470d0a1a0a'))
b'\x89PNG\r\n\x1a\n'
```

7.2.5　位运算符

Python 提供了和 C 语言中类似的比特级运算符。表 7-7 列出了这些位运算符并附上了整数 a（十进制 5，二进制 0b0101）和 b（十进制 1，二进制 0b0001）的运算示例。

表7-7：比特级整数运算符

运算符	描述	示例	十进制结果	二进制结果
&	与	a & b	1	0b0001
|	或	a | b	5	0b0101
^	异或	a ^ b	4	0b0100
~	翻转	~a	-6	取决于 int 类型的大小
<<	左位移	a << 1	10	0b1010
>>	右位移	a >> 1	2	0b0010

这些运算和第 3 章的集合运算有些类似。& 返回两个运算数中相同的比特。| 返回两个运算数中任意一者有效的比特。^ 返回仅在一个运算数中有效的比特。~ 将所有比特翻转。现代计算机都使用*二进制补码*（two’s complement）进行运算，其中整数的最高位定义为符号位（0 为正，1 为负），因此翻转操作会改变运算数的符号。<< 和 >> 仅仅将比特向左或向右移动，左位移操作相当于将数字乘以 2，右位移操作相当于将数字除以 2。

7.3　练习

(1) 创建一个 Unicode 字符串 mystery 并将它的值设为 '\U0001f4a9'。打印 mystery，并查看 mystery 的 Unicode 名称。

(2) 使用 UTF-8 对 mystery 进行编码，存入字节型变量 pop_bytes，并将它打印出来。

(3) 使用 UTF-8 对 `pop_bytes` 进行解码，存入字符串型变量 `pop_string`，并将它打印出来，看看它与 `mystery` 是否一致？

(4) 使用旧式格式化方法生成下面的诗句，把 `'roast beef'`、`'ham'`、`'head'` 和 `'clam'` 依次插入字符串：

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.
```

(5) 使用新式格式化方法生成下面的套用信函，将下面的字符串存储为 `letter`（后面的练习中会用到）：

```
Dear {salutation} {name},

Thank you for your letter. We are sorry that our {product} {verbed} in your
{room}. Please note that it should never be used in a {room}, especially
near any {animals}.

Send us your receipt and {amount} for shipping and handling. We will send
you another {product} that, in our tests, is {percent}% less likely to
have {verbed}.

Thank you for your support.

Sincerely,
{spokesman}
{job_title}
```

(6) 创建一个字典 `response` 包含以下键：`'salutaion'`、`'name'`、`'product'`、`'verved'`（动词过去式）、`'room'`、`'animals'`、`'amount'`、`'percent'`、`'spokesman'` 以及 `'job_title'`。设定这些键对应的值，并打印由 `response` 的值填充的 `letter`。

(7) 正则表达式在处理文本上非常方便，在这个练习中我们会对示例文本尝试做各种各样的操作。示例文本是一首诗，名为 *Ode on the Mammoth Cheese*，作者是 James McIntyre，写于 1866 年。出于对当时安大略湖手工制造的 7000 磅的巨型奶酪的敬意，它当时甚至在全球巡回展出。如果你不愿意自己一词一句敲出来，直接百度一下粘贴到你的 Python 代码里即可。你也可以从 Project Gutenberg（http://www.gutenberg.org/ebooks/36068?msg=welcome_stranger）找到。我们将这个字符串命名为 `mammoth`。

```
We have seen thee, queen of cheese,
Lying quietly at your ease,
Gently fanned by evening breeze,
Thy fair form no flies dare seize.

All gaily dressed soon you'll go
To the great Provincial show,
To be admired by many a beau
In the city of Toronto.

Cows numerous as a swarm of bees,
Or as the leaves upon the trees,
```

```
It did require to make thee please,
And stand unrivalled, queen of cheese.

May you not receive a scar as
We have heard that Mr. Harris
Intends to send you off as far as
The great world's show at Paris.

Of the youth beware of these,
For some of them might rudely squeeze
And bite your cheek, then songs or glees
We could not sing, oh! queen of cheese.

We'rt thou suspended from balloon,
You'd cast a shade even at noon,
Folks would think it was the moon
About to fall and crush them soon.
```

(8) 引入 `re` 模块以便使用正则表达式相关函数。使用 `re.findall()` 打印出所有以 `c` 开头的单词。

(9) 找到所有以 `c` 开头的 4 个字母的单词。

(10) 找到所有以 `r` 结尾的单词。

(11) 找到所有包含且仅包含 3 个元音的单词。

(12) 使用 `unhexlify` 将下面的十六进制串（出于排版原因将它们拆成两行字符串）转换为 `bytes` 型变量，命名为 `gif`：

```
'47494638396101000100800000000000ffffff21f9' +
'0401000000002c000000000100010000020144003b'
```

(13) `gif` 定义了一个 1 像素的透明 GIF 文件（最常见的图片格式之一）。合法的 GIF 文件开头由 GIF89a 组成，检测一下上面的 `gif` 是否为合法的 GIF 文件？

(14) GIF 文件的像素宽度是一个 16 比特的以大端方案存储的整数，偏移量为 6 字节，高度数据的大小与之相同，偏移量为 8。从 `gif` 中抽取这些信息并打印出来，看看它们是否与预期的一样都为 1？

第 8 章

数据的归宿

“在没有事实（数据）作为参考的情况下妄下结论是个可怕的错误。”

——阿瑟 · 柯南 · 道尔

一个运行中的程序会存取放在随机存取存储器（RAM）上的数据。RAM 读取速度快，但价格昂贵，需要持续供电，断电后保存在上面的数据会自动消失。磁盘速度比 RAM 慢，但容量大、费用低廉并且多次插拔电源线仍可保持数据。因此，计算机系统在数据存储设计中做出很大的努力来权衡磁盘和 RAM。程序员需要在非易失性介质（例如磁盘）上做持久化存储和检索数据。

本章会涉及不同类型的数据存储，它们基于不同的目的进行优化：普通文件、结构化文件和数据库。除了输入和输出，文件操作都会在 10.1 节讲到。

> 本章也是第一次讲到非标准的 Python 模块，也就是除了标准库之外的 Python 代码。你可以通过 `pip` 命令轻松地安装这些第三方库，更多的使用细节详见附录 D。

8.1 文件输入/输出

数据持久化最简单的类型是普通文件，有时也叫平面文件（flat file）。它仅仅是在一个文件名下的字节流，把数据从一个文件读入内存，然后从内存写入文件。Python 很容易实现这些文件操作，它模仿熟悉的和流行的 Unix 系统的操作。

读写一个文件之前需要打开它：

```
fileobj = open(filename, mode)
```

下面是对该 open() 调用的简单解释：

- fileobj 是 open() 返回的文件对象；
- filename 是该文件的字符串名；
- mode 是指明文件类型和操作的字符串。

mode 的第一个字母表明对其的操作。

- r 表示读模式。
- w 表示写模式。如果文件不存在则新创建，如果存在则重写新内容。
- x 表示在文件不存在的情况下新创建并写文件。
- a 表示如果文件存在，在文件末尾追加写内容。

mode 的第二个字母是文件类型：

- t（或者省略）代表文本类型；
- b 代表二进制文件。

打开文件之后就可以调用函数来读写数据，之后的例子会涉及。

最后需要关闭文件。

接下来在一个程序中用 Python 字符串创建一个文件，然后返回。

8.1.1　使用write()写文本文件

出于一些原因，我们没有太多的关于狭义相对论的五行打油诗（limerick[1]）。下面这首作为源数据：

```
>>> poem = '''There was a young lady named Bright,
... Whose speed was far faster than light;
... She started one day
... In a relative way,
... And returned on the previous night.'''
>>> len(poem)
150
```

以下代码将整首诗写到文件 'relativity' 中：

```
>>> fout = open('relativity', 'wt')
>>> fout.write(poem)
150
>>> fout.close()
```

函数 write() 返回写入文件的字节数。和 print() 一样，它没有增加空格或者换行符。同样，你也可以在一个文本文件中使用 print()：

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout)
>>> fout.close()
```

注 1：一种通俗幽默的短诗。——译者注

这就产生了一个问题：到底是使用 write() 还是 print()？ print() 默认会在每个参数后面添加空格，在每行结束处添加换行。 在之前的例子中，它在文件 relativity 中默认添加了一个换行。为了使 print() 与 write() 有同样的输出，传入下面两个参数。

- sep 分隔符：默认是一个空格 ' '
- end 结束字符：默认是一个换行符 '\n'

除非自定义参数，否则 print() 会使用默认参数。在这里，我们通过空字符串替换 print() 添加的所有多余输出：

```
>>> fout = open('relativity', 'wt')
>>> print(poem, file=fout, sep='', end='')
>>> fout.close()
```

如果源字符串非常大，可以将数据分块，直到所有字符被写入：

```
>>> fout = open('relativity', 'wt')
>>> size = len(poem)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...          break
...     fout.write(poem[offset:offset+chunk])
...     offset += chunk
...
100
50
>>> fout.close()
```

第一次写入 100 个字符，然后写入剩下的 50 个字符。

如果 'relativity' 文件已经存在，使用模式 x 可以避免重写文件：

```
>>> fout = open('relativity', 'xt')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileExistsError: [Errno 17] File exists: 'relativity'
```

可以加入一个异常处理：

```
>>> try:
...     fout = open('relativity', 'xt')
...     fout.write('stomp stomp stomp')
... except FileExistsError:
...     print('relativity already exists!. That was a close one.')
...
relativity already exists!. That was a close one.
```

8.1.2　使用read()、readline()或者readlines()读文本文件

你可以按照下面的示例那样，使用不带参数的 read() 函数一次读入文件的所有内容。但在读入文件时要格外注意，1 GB 的文件会用到相同大小的内存。

```
>>> fin = open('relativity', 'rt' )
>>> poem = fin.read()
>>> fin.close()
>>> len(poem)
150
```

同样也可以设置最大的读入字符数限制 read() 函数一次返回的大小。下面一次读入 100 个字符，然后把每一块拼接成原来的字符串 poem：

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> chunk = 100
>>> while True:
...     fragment = fin.read(chunk)
...     if not fragment:
...         break
...     poem += fragment
...
>>> fin.close()
>>> len(poem)
150
```

读到文件结尾之后，再次调用 read() 会返回空字符串（''），if not fragment 条件被判为 False。此时会跳出 while True 的循环。 当然，你也能使用 readline() 每次读入文件的一行。在下一个例子中，通过追加每一行拼接成原来的字符串 poem：

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> while True:
...     line = fin.readline()
...     if not line:
...         break
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

对于一个文本文件，即使空行也有 1 字符长度（换行字符 '\n'），自然就会返回 True。当文件读取结束后，readline()（类似 read()）同样会返回空字符串，也被 while True 判为 False。

读取文本文件最简单的方式是使用一个迭代器（iterator），它会每次返回一行。这和之前的例子类似，但代码会更短：

```
>>> poem = ''
>>> fin = open('relativity', 'rt' )
>>> for line in fin:
...     poem += line
...
>>> fin.close()
>>> len(poem)
150
```

前面所有的示例最终都返回单个字符串 poem。函数 readlines() 调用时每次读取一行，并返回单行字符串的列表：

```
>>> fin = open('relativity', 'rt' )
>>> lines = fin.readlines()
>>> fin.close()
>>> print(len(lines), 'lines read')
5 lines read
>>> for line in lines:
...     print(line, end='')
...
There was a young lady named Bright,
Whose speed was far faster than light;
She started one day
In a relative way,
And returned on the previous night.>>>
```

之前我们让 print() 去掉每行结束的自动换行，因为前面的四行都有换行标志，而最后一行没有，所以导致解释器的提示符 >>> 出现在最后一行的最右边。

8.1.3　使用write()写二进制文件

如果文件模式字符串中包含 'b'，那么文件会以二进制模式打开。这种情况下，读写的是字节而不是字符串。

我们手边没有二进制格式的诗，所以直接在 0~255 产生 256 字节的值：

```
>>> bdata = bytes(range(0, 256))
>>> len(bdata)
256
```

以二进制模式打开文件，并且一次写入所有的数据：

```
>>> fout = open('bfile', 'wb')
>>> fout.write(bdata)
256
>>> fout.close()
```

再次，write() 返回到写入的字节数。

对于文本，也可以分块写二进制数据：

```
>>> fout = open('bfile', 'wb')
>>> size = len(bdata)
>>> offset = 0
>>> chunk = 100
>>> while True:
...     if offset > size:
...          break
...     fout.write(bdata[offset:offset+chunk])
...     offset += chunk
...
100
100
56
>>> fout.close()
```

8.1.4 使用read()读二进制文件

下面简单的例子只需要用 'rb' 打开文件即可：

```
>>> fin = open('bfile', 'rb')
>>> bdata = fin.read()
>>> len(bdata)
256
>>> fin.close()
```

8.1.5 使用with自动关闭文件

如果你忘记关闭已经打开的一个文件，在该文件对象不再被引用之后 Python 会关掉此文件。这也就意味着在一个函数中打开文件，没有及时关闭它，但是在函数结束时会被关掉。然而你可能会在一直运行中的函数或者程序的主要部分打开一个文件，应该强制剩下的所有写操作完成后再关闭文件。

Python 的上下文管理器（context manager）会清理一些资源，例如打开的文件。它的形式为 with expression as variable：

```
>>> with open('relativity', 'wt') as fout:
...     fout.write(poem)
...
```

完成上下文管理器的代码后，文件会被自动关闭。

8.1.6 使用seek()改变位置

无论是读或者写文件，Python 都会跟踪文件中的位置。函数 tell() 返回距离文件开始处的字节偏移量。函数 seek() 允许跳转到文件其他字节偏移量的位置。这意味着可以不用从头读取文件的每一个字节，直接跳到最后位置并只读一个字节也是可行的。

对于这个例子，使用之前写过的 256 字节的二进制文件 'bfile'：

```
>>> fin = open('bfile', 'rb')
>>> fin.tell()
0
```

使用 seek() 读取文件结束前最后一个字节：

```
>>> fin.seek(255)
255
```

一直读到文件结束：

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

seek() 同样返回当前的偏移量。

用第二个参数调用函数 seek()：seek(offset,origin)。

- 如果 origin 等于 0（默认为 0），从开头偏移 offset 个字节；
- 如果 origin 等于 1，从当前位置处偏移 offset 个字节；
- 如果 origin 等于 2，距离最后结尾处偏移 offset 个字节。

这些值也在标准 os 模块中被定义：

```
>>> import os
>>> os.SEEK_SET
0
>>> os.SEEK_CUR
1
>>> os.SEEK_END
2
```

所以，我们可以用不同的方法读取最后一个字节：

```
>>> fin = open('bfile', 'rb')
```

文件结尾前的一个字节：

```
>>> fin.seek(-1, 2)
255
>>> fin.tell()
255
```

一直读到文件结尾：

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

在调用 seek() 函数时不需要额外调用 tell()。前面的例子只是想说明两个函数都可以返回同样的偏移量。

下面是从文件的当前位置寻找的例子：

```
>>> fin = open('bfile', 'rb')
```

接下来的例子返回最后两个字节：

```
>>> fin.seek(254, 0)
254
>>> fin.tell()
254
```

在此基础上前进一个字节：

```
>>> fin.seek(1, 1)
255
>>> fin.tell()
255
```

最后一直读到文件结尾：

```
>>> bdata = fin.read()
>>> len(bdata)
1
>>> bdata[0]
255
```

这些函数对于二进制文件都是极其重要的。当文件是 ASCII 编码（每个字符一个字节）时，也可以使用它们，但是计算偏移量会是一件麻烦事。其实，这些都取决于文本的编码格式，最流行的编码格式（例如 UTF-8）每个字符的字节数都不尽相同。

8.2 结构化的文本文件

对于简单的文本文件，唯一的结构层次是间隔的行。然而有时候需要更加结构化的文本，用于后续使用的程序保存数据或者向另外一个程序传送数据。

结构化的文本有很多格式，区别它们的方法如下所示。

- 分隔符，比如 tab（`'\t'`）、逗号（`','`）或者竖线（`'|'`）。逗号分隔值（CSV）就是这样的例子。
- `'<'` 和 `'>'` 标签，例如 XML 和 HTML。
- 标点符号，例如 JavaScript Object Notation（JSON[2]）。
- 缩进，例如 YAML（即 YAML Ain't Markup Language 的缩写），要了解更多可以去搜索。
- 混合的，例如各种配置文件。

每一种结构化文件格式都能够被至少一种 Python 模块读写。

8.2.1 CSV

带分隔符的文件一般用作数据交换格式或者数据库。你可以人工读入 CSV 文件，每一次读取一行，在逗号分隔符处将每行分开，并添加结果到某些数据结构中，例如列表或者字典。但是，最好使用标准的 `csv` 模块，因为这样切分会得到更加复杂的信息。

- 除了逗号，还有其他可代替的分隔符：`'|'` 和 `'\t'` 很常见。
- 有些数据会有转义字符序列，如果分隔符出现在一块区域内，则整块都要加上引号或者在它之前加上转义字符。
- 文件可能有不同的换行符，Unix 系统的文件使用 `'\n'`，Microsoft 使用 `'\r\n'`，Apple 之前使用 `'\r'` 而现在使用 `'\n'`。
- 在第一行可以加上列名。

注 2：JSON 是一种轻量级的数据交换格式，它是基于 JavaScript 的一个子集。——译者注

首先读和写一个列表的行，每一行包含很多列：

```
>>> import csv
>>> villains = [
...     ['Doctor', 'No'],
...     ['Rosa', 'Klebb'],
...     ['Mister', 'Big'],
...     ['Auric', 'Goldfinger'],
...     ['Ernst', 'Blofeld'],
...     ]
>>> with open('villains', 'wt') as fout:  # 一个上下文管理器
...     csvout = csv.writer(fout)
...     csvout.writerows(villains)
```

于是创建了包含以下几行的文件 villains：

```
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

现在，我们来重新读这个文件：

```
>>> import csv
>>> with open('villains', 'rt') as fin:  # 一个上下文管理器
...     cin = csv.reader(fin)
...     villains = [row for row in cin]  # 使用列表推导式
...
>>> print(villains)
[['Doctor', 'No'], ['Rosa', 'Klebb'], ['Mister', 'Big'],
['Auric', 'Goldfinger'], ['Ernst', 'Blofeld']]
```

停下来想想列表推导式（随时打开 4.6 节，温习一下它的语法）。我们利用函数 `reader()` 创建的结构，它在通过 `for` 循环提取到的 `cin` 对象中构建每一行。

使用 `reader()` 和 `writer()` 的默认操作。每一列用逗号分开；每一行用换行符分开。

数据可以是字典的集合（a list of dictionary），不仅仅是列表的集合（a list of list）。这次使用新函数 `DictReader()` 读取文件 villains，并且指定每一列的名字：

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin, fieldnames=['first', 'last'])
...     villains = [row for row in cin]
...
>>> print(villains)
[{'last': 'No', 'first': 'Doctor'},
{'last': 'Klebb', 'first': 'Rosa'},
{'last': 'Big', 'first': 'Mister'},
{'last': 'Goldfinger', 'first': 'Auric'},
{'last': 'Blofeld', 'first': 'Ernst'}]
```

下面使用新函数 `DictWriter()` 重写 CSV 文件，同时调用 `writeheader()` 向 CSV 文件中第一行写入每一列的名字：

```
import csv
villains = [
    {'first': 'Doctor', 'last': 'No'},
    {'first': 'Rosa', 'last': 'Klebb'},
    {'first': 'Mister', 'last': 'Big'},
    {'first': 'Auric', 'last': 'Goldfinger'},
    {'first': 'Ernst', 'last': 'Blofeld'},
    ]
with open('villains', 'wt') as fout:
    cout = csv.DictWriter(fout, ['first', 'last'])
    cout.writeheader()
    cout.writerows(villains)
```

于是创建了具有标题行的新文件 villains：

```
first,last
Doctor,No
Rosa,Klebb
Mister,Big
Auric,Goldfinger
Ernst,Blofeld
```

回过来再读取写入的文件，忽略函数 `DictReader()` 调用的参数 `fieldnames`，把第一行的值（`first,last`）作为列标签，和字典的键做匹配：

```
>>> import csv
>>> with open('villains', 'rt') as fin:
...     cin = csv.DictReader(fin)
...     villains = [row for row in cin]
...
>>> print(villains)
[{'last': 'No', 'first': 'Doctor'},
{'last': 'Klebb', 'first': 'Rosa'},
{'last': 'Big', 'first': 'Mister'},
{'last': 'Goldfinger', 'first': 'Auric'},
{'last': 'Blofeld', 'first': 'Ernst'}]
```

8.2.2 XML

带分隔符的文件仅有两维的数据：行和列。如果你想在程序之间交换数据结构，需要一种方法把层次结构、序列、集合和其他的结构编码成文本。

XML 是最突出的处理这种转换的标记（markup）格式，它使用标签（tag）分隔数据，如下面的示例文件 menu.xml 所示：

```
<?xml version="1.0"?>
<menu>
  <breakfast hours="7-11">
    <item price="$6.00">breakfast burritos</item>
    <item price="$4.00">pancakes</item>
  </breakfast>
  <lunch hours="11-3">
    <item price="$5.00">hamburger</item>
  </lunch>
```

```
  <dinner hours="3-10">
    <item price="8.00">spaghetti</item>
  </dinner>
</menu>
```

以下是 XML 的一些重要特性：

- 标签以一个 < 字符开头，例如示例中的标签 menu、breakfast、lunch、dinner 和 item；
- 忽略空格；
- 通常一个开始标签（例如 <menu>）跟一段其他的内容，然后是最后相匹配的结束标签，例如 </menu>；
- 标签之间是可以存在多级嵌套的，在本例中，标签 item 是标签 breakfast、lunch 和 dinner 的子标签，反过来，它们也是标签 menu 的子标签；
- 可选属性（attribute）可以出现在开始标签里，例如 price 是 item 的一个属性；
- 标签中可以包含值（value），本例中每个 item 都会有一个值，比如第二个 breakfast item 的 pancakes；
- 如果一个命名为 thing 的标签没有内容或者子标签，它可以用一个在右尖括号的前面添加斜杠的简单标签所表示，例如 <thing/> 代替开始和结束都存在的标签 <thing> 和 </thing>；
- 存放数据的位置可以是任意的——属性、值或者子标签。例如也可以把最后一个 item 标签写作 <item price ="$8.00" food ="spaghetti"/>。

XML 通常用于数据传送和消息，它存在一些子格式，如 RSS 和 Atom。工业界有许多定制化的 XML 格式，例如金融领域（http://www.service-architecture.com/articles/xml/finance_xml.html）。

XML 的灵活性导致出现了很多方法和性能各异的 Python 库。

在 Python 中解析 XML 最简单的方法是使用 ElementTree，下面的代码用来解析 menu.xml 文件以及输出一些标签和属性：

```
>>> import xml.etree.ElementTree as et
>>> tree = et.ElementTree(file='menu.xml')
>>> root = tree.getroot()
>>> root.tag
'menu'
>>> for child in root:
...     print('tag:', child.tag, 'attributes:', child.attrib)
...     for grandchild in child:
...         print('\ttag:', grandchild.tag, 'attributes:', grandchild.attrib)
...
tag: breakfast attributes: {'hours': '7-11'}
    tag: item attributes: {'price': '$6.00'}
    tag: item attributes: {'price': '$4.00'}
tag: lunch attributes: {'hours': '11-3'}
    tag: item attributes: {'price': '$5.00'}
tag: dinner attributes: {'hours': '3-10'}
    tag: item attributes: {'price': '8.00'}
>>> len(root)     # 菜单选择的数目
3
```

```
>>> len(root[0])  # 早餐项的数目
2
```

对于嵌套列表中的每一个元素，`tag` 是标签字符串，`attrib` 是它属性的一个字典。`ElementTree` 有许多查找 XML 导出数据、修改数据乃至写入 XML 文件的方法，它的文档（https://docs.python.org/3.3/library/xml.etree.elementtree.html）中有详细的介绍。

其他标准的 Python XML 库如下。

- `xml.dom`
  JavaScript 开发者比较熟悉的文档对象模型（DOM）将 Web 文档表示成层次结构，它会把整个 XML 文件载入到内存中，同样允许你获取所有的内容。
- `xml.sax`
  简单的 XML API 或者 SAX 都是通过在线解析 XML，不需要一次载入所有内容到内存中，因此对于处理巨大的 XML 文件流是一个很好的选择。

8.2.3 HTML

在 Web 网络中，海量的数据以超文本标记语言（HTML）这一基本的文档格式存储。然而许多文档不遵循 HTML 的规则，导致很难进行解析。况且更多的 HTML 是用来格式化输出显示结果而不是用于交换数据。本章的主要内容是描述定义明确的数据格式，关于 HTML 的更多讨论放在第 9 章。

8.2.4 JSON

JavaScript Object Notation（JSON，http://www.json.org）是源于 JavaScript 的当今很流行的数据交换格式，它是 JavaScript 语言的一个子集，也是 Python 合法可支持的语法。对于 Python 的兼容性使得它成为程序间数据交换的较好选择。同样，在第 9 章的 Web 开发中会看到很多 JSON 的示例。

不同于众多的 XML 模块，Python 只有一个主要的 JSON 模块 `json`（名字容易记忆）。下面的程序将数据编码成 JSON 字符串，然后再把 JSON 字符串解码成数据。用之前 XML 例子的数据构建 JSON 的数据结构：

```
>>> menu = \
... {
... "breakfast": {
...         "hours": "7-11",
...         "items": {
...                 "breakfast burritos": "$6.00",
...                 "pancakes": "$4.00"
...                 }
...         },
... "lunch" : {
...         "hours": "11-3",
...         "items": {
...                 "hamburger": "$5.00"
...                 }
...         },
```

```
... "dinner": {
...         "hours": "3-10",
...         "items": {
...                 "spaghetti": "$8.00"
...                 }
...         }
... }
.
```

接下来使用 `dumps()` 将 `menu` 编码成 JSON 字符串（`menu_json`）：

```
>>> import json
>>> menu_json = json.dumps(menu)
>>> menu_json
'{"dinner": {"items": {"spaghetti": "$8.00"}, "hours": "3-10"},
"lunch": {"items": {"hamburger": "$5.00"}, "hours": "11-3"},
"breakfast": {"items": {"breakfast burritos": "$6.00", "pancakes":
"$4.00"}, "hours": "7-11"}}'
```

现在反过来使用 `loads()` 把 JSON 字符串 `menu_json` 解析成 Python 的数据结构（`menu2`）：

```
>>> menu2 = json.loads(menu_json)
>>> menu2
{'breakfast': {'items': {'breakfast burritos': '$6.00', 'pancakes':
'$4.00'}, 'hours': '7-11'}, 'lunch': {'items': {'hamburger': '$5.00'},
'hours': '11-3'}, 'dinner': {'items': {'spaghetti': '$8.00'}, 'hours': '3-10'}}
```

`menu` 和 `menu2` 是具有相同键值的字典，和标准的字典用法一样，得到的键的顺序是不尽相同的。

你可能会在编码或者解析 JSON 对象时得到异常，包括对象的时间 `datetime`（在 10.4 节详细介绍）：

```
>>> import datetime
>>> now = datetime.datetime.utcnow()
>>> now
datetime.datetime(2013, 2, 22, 3, 49, 27, 483336)
>>> json.dumps(now)
Traceback (most recent call last):
# …… (删除栈跟踪以保存树)
TypeError: datetime.datetime(2013, 2, 22, 3, 49, 27, 483336) is not JSON serializable
>>>
```

上述错误发生是因为标准 JSON 没有定义日期或者时间类型，需要自定义处理方式。你可以把 `datetime` 转换成 JSON 能够理解的类型，比如字符串或者 epoch 值（第 10 章讲解）：

```
>>> now_str = str(now)
>>> json.dumps(now_str)
'"2013-02-22 03:49:27.483336"'
>>> from time import mktime
>>> now_epoch = int(mktime(now.timetuple()))
>>> json.dumps(now_epoch)
'1361526567'
```

如果 `datetime` 值出现在正常转换后的数据中间，那么做上面的特殊转化是困难的。你可

以通过继承修改 JSON 的编码方式（6.3 节中讲解），Python 中的 JSON 文档（https://docs.python.org/3.3/library/json.html）给出了一个复杂数字的例子，同样使 JSON 出现问题。下面为 `datetime` 修改编码方式：

```
>>> class DTEncoder(json.JSONEncoder):
...     def default(self, obj):
...         # isinstance()检查obj的类型
...         if isinstance(obj, datetime.datetime):
...             return int(mktime(obj.timetuple()))
...         # 否则是普通解码器知道的东西：
...         return json.JSONEncoder.default(self, obj)
...
>>> json.dumps(now, cls=DTEncoder)
'1361526567'
```

新类 `DTEncoder` 是 `JSONEncoder` 的一个子类。我们需要重载它的 `default()` 方法来增加处理 `datetime` 的代码。继承确保了剩下的功能与父类的一致性。

函数 `isinstance()` 检查 `obj` 是否是类 `datetime.datetime` 的对象，因为在 Python 中一切都是对象，`isinstance()` 总是适用的：

```
>>> type(now)
<class 'datetime.datetime'>
>>> isinstance(now, datetime.datetime)
True
>>> type(234)
<class 'int'>
>>> isinstance(234, int)
True
>>> type('hey')
<class 'str'>
>>> isinstance('hey', str)
True
```

对于 JSON 和其他结构化的文本格式，在不需要提前知道任何东西的情况下可以从一个文件中解析数据结构。然后使用 `isinstance()` 和相关类型的方法遍历数据。例如，如果其中的一项是字典结构，可以通过 `keys()`、`values()` 和 `items()` 抽取内容。

8.2.5　YAML

和 JSON 类似，YAML（http://www.yaml.org）同样有键和值，但主要用来处理日期和时间这样的数据类型。标准的 Python 库没有处理 YAML 的模块，因此需要安装第三方库 `yaml`（http://pyyaml.org/wiki/PyYAML）操作数据。`load()` 将 YAML 字符串转换为 Python 数据结构，而 `dump()` 正好相反。

下面的 YAML 文件 mcintyre.yaml 包含加拿大诗人 James McIntyre 的两首诗的信息：

```
name:
  first: James
  last: McIntyre
```

```
dates:
  birth: 1828-05-25
  death: 1906-03-31
details:
  bearded: true
  themes: [cheese, Canada]
books:
  url: http://www.gutenberg.org/files/36068/36068-h/36068-h.htm
poems:
  - title: 'Motto'
    text: |
      Politeness, perseverance and pluck,
      To their possessor will bring good luck.
  - title: 'Canadian Charms'
    text: |
      Here industry is not in vain,
      For we have bounteous crops of grain,
      And you behold on every field
      Of grass and roots abundant yield,
      But after all the greatest charm
      Is the snug home upon the farm,
      And stone walls now keep cattle warm.
```

类似于 `true`、`false`、`on` 和 `off` 的值可以转换为 Python 的布尔值。整数和字符串转换为 Python 等价的。其他语法创建列表和字典：

```
>>> import yaml
>>> with open('mcintyre.yaml', 'rt') as fin:
>>>     text = fin.read()
>>> data = yaml.load(text)
>>> data['details']
{'themes': ['cheese', 'Canada'], 'bearded': True}
>>> len(data['poems'])
2
```

创建的匹配这个 YAML 文件的数据结构超过了一层嵌套。如果你想得到第二首诗歌的题目，使用 dict/list/dict 的引用：

```
>>> data['poems'][1]['title']
'Canadian Charms'
```

> PyYAML 可以从字符串中载入 Python 对象，但这样做是不安全的。如果导入你不信任的 YAML，使用 `safe_load()` 代替 `load()`。最好还是使用 `safe_load()`，通过阅读 war is peace（http://nedbatchelder.com/blog/201302/war_is_peace.html）进一步了解载入 YAML 在 Ruby on Rails 平台上如何折中。

8.2.6 安全提示

你可以使用本章中介绍的所有格式保存数据对象到文件中，或者在文件中读取它们。在这个过程中也可能会产生安全性问题。

例如，下面引自 10 亿维基百科页面的 XML 片段定义了 10 个嵌套实体，每一项扩展 10 倍

的子项，总共有 10 亿的扩展项：

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
 <!ENTITY lol "lol">
 <!ENTITY lol1 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
 <!ENTITY lol2 "&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;&lol1;">
 <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
 <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
 <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
 <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
 <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
 <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
 <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

糟糕的是，前面提到的 XML 库无法容纳 10 亿多的项。Defused XML（https://bitbucket.org/tiran/defusedxml）列出了这种攻击和 Python 库的其他缺点，并且指出了如何修改设置避免这些问题。或者使用 `defusedxml` 库作为安全的保护：

```
>>> # 不安全：
>>> from xml.etree.ElementTree import parse
>>> et = parse(xmlfile)
>>> # 受保护：
>>> from defusedxml.ElementTree import parse
>>> et = parse(xmlfile)
```

8.2.7　配置文件

许多程序提供多种*选项*和*设置*。动态的设置可以通过传入程序参数，但是持久的参数需要保存下来。因此，尽管你急着想快速定义自己的*配置文件*格式，但是最好不要这样做。你定义的格式可能既粗糙也没有节省时间。你需要同时维护写入配置文件的程序以及读取配置文件的程序（有时被称为*解析程序*）。其实，在程序中配置文件可以有许多好的选择，包括之前几节中提到的格式。

我们使用标准 `configparser` 模块处理 Windows 风格的初始化 .ini 文件。这些文件都包含 key = value 的定义，以下是一个简单的配置文件 settings.cfg 例子：

```
[english]
greeting = Hello

[french]
greeting = Bonjour

[files]
home = /usr/local
# 简单的插入：
bin = %(home)s/bin
```

下面的代码是把配置文件读入到 Python 数据结构：

```
>>> import configparser
>>> cfg = configparser.ConfigParser()
>>> cfg.read('settings.cfg')
['settings.cfg']
>>> cfg
<configparser.ConfigParser object at 0x1006be4d0>
>>> cfg['french']
<Section: french>
>>> cfg['french']['greeting']
'Bonjour'
>>> cfg['files']['bin']
'/usr/local/bin'
```

其他操作包括自定义修改也是可以实现的，请参阅文档 configparser（https://docs.python.org/3.3/library/configparser.html）。如果你需要两层以上的嵌套结构，使用 YAML 或者 JSON。

8.2.8　其他交换格式

这些二进制数据交换格式通常比 XML 或者 JSON 更加快速和复杂：

- MsgPack（http://msgpack.org）
- Protocol Buffers（https://code.google.com/p/protobuf/）
- Avro（http://avro.apache.org/docs/current/）
- Thrift（http://thrift.apache.org/）

因为它们都是二进制文件，所以人类是无法使用文本编辑器轻易编辑的。

8.2.9　使用pickle序列化

存储数据结构到一个文件中也称为*序列化*（serializing）。像 JSON 这样的格式需要定制的序列化数据的转换器。Python 提供了 pickle 模块以特殊的二进制格式保存和恢复数据对象。

还记得 JSON 解析 datetime 对象时出现问题吗？但对于 pickle 就不存在问题：

```
>>> import pickle
>>> import datetime
>>> now1 = datetime.datetime.utcnow()
>>> pickled = pickle.dumps(now1)
>>> now2 = pickle.loads(pickled)
>>> now1
datetime.datetime(2014, 6, 22, 23, 24, 19, 195722)
>>> now2
datetime.datetime(2014, 6, 22, 23, 24, 19, 195722)
```

pickle 同样也适用于自己定义的类和对象。现在，我们定义一个简单的类 Tiny，当其对象强制转换为字符串时会返回 'tiny'：

```
>>> import pickle
>>> class Tiny():
...     def __str__(self):
```

```
...         return 'tiny'
...
>>> obj1 = Tiny()
>>> obj1
<__main__.Tiny object at 0x10076ed10>
>>> str(obj1)
'tiny'
>>> pickled = pickle.dumps(obj1)
>>> pickled
b'\x80\x03c__main__\nTiny\nq\x00)\x81q\x01.'
>>> obj2 = pickle.loads(pickled)
>>> obj2
<__main__.Tiny object at 0x10076e550>
>>> str(obj2)
'tiny'
```

`pickled` 是从对象 `obj1` 转换来的序列化二进制字符串。然后再把字符串还原成对象 `obj1` 的副本 `obj2`。使用函数 `dump()` 序列化数据到文件，而函数 `load()` 用作反序列化。

> 因为 `pickle` 会创建 Python 对象，前面提到的安全问题也同样会发生，不要对你不信任的文件做反序列化。

8.3　结构化二进制文件

有些文件格式是为了存储特殊的数据结构，它们既不是关系型数据库也不是 NoSQL 数据库。下面会介绍其中的几种。

8.3.1　电子数据表

电子数据表，尤其是 Microsoft Excel，是广泛使用的二进制数据格式。如果你把电子数据表保存到一个 CSV 文件中，就可以利用之前提到的标准 `csv` 模块读取它。如果你有一个 xls 文件，也可以使用第三方库 `xlrd`（http://pypi.python.org/pypi/xlrd）读写文件。

8.3.2　层次数据格式

层次数据格式（HDF5）是一种用于多维数据或者层次数值数据的二进制数据格式。它主要用在科学计算领域，快速读取海量数据集（GB 或者 TB）是常见的需求。即使某些情况下 HDF5 能很好地代替数据库，但它在商业应用上也是默默无闻的。它能适用于 WORM（Write Once/Read Many；一次写入，多次读取）应用，不用担心写操作冲突的数据保护。下面是两个可能有用的模块：

- `h5py` 是功能全面的较低级的接口，参考文档（http://www.h5py.org/）和代码（https://github.com/h5py/h5py）；
- `PyTables` 是具有数据库特征的较为高级的接口，参考文档（http://www.pytables.org/）和代码（http://pytables.github.com/）。

这两个模块都会在附录 C 中的 Python 科学应用中讨论，在这里提到它以防你有存储和检索海量大数据的需求以及愿意考虑不同于普通数据库的解决方案。一个很好的例子是 Million Song dataset（http://labrosa.ee.columbia.edu/millionsong/），就是通过 HDF5 格式下载歌曲数据。

8.4 关系型数据库

关系型数据库虽然只有 40 多年的历史，却无处不在。你一定曾经和它打过交道，使用时你会体会到它提供的如下功能：

- 多用户同时访问数据；
- 用户使用数据的保护；
- 高效地存储和检索数据；
- 数据被*模式*定义以及被*约束*限制；
- Joins 通过连接发现不同数据之间的关系；
- 声明式（非命令式）查询语言，SQL（Structured Query Language）。

之所以被称为*关系型*（relational）是因为数据库展现了*表单*（table）形式的不同类型数据之间的关系。例如之前菜单的例子中，每一项和它的价格是有对应关系的。

表单是一个具有行和列的二元组，和电子数据表类似。要创建一个表单，需要给它命名，明确次序、每一项的名称以及每一列的类型。每一行都会存在相同的列，即使允许缺失项（也称为 null）。在菜单例子中，你创建的表单中应该把每一个待售的食物作为一行；每一行都存在相同的列，包括它的价格。

某一行或者某几行通常作为表单的*主键*，在表单中主键的值是独一无二的，防止重复增添数据项。这些键在查询时被快速索引，类似于图书的索引，方便快速地找到指定行。

每一个表单都附属于某数据库，类似于一个文件都存在于某目录下。两层的层次结构便于更好地组织和管理。

> 数据库一词有多种用法，用于指代服务器、表单容器以及存储的数据。如果你同时指代它们，可以称其为*数据库服务器*（database server）、*数据库*（database）和*数据*（data）。

如果你想通过非主键的列的值查找数据，可以定义一个*二级索引*，否则数据库服务器需要扫描整个表单，暴力搜索每一行找到匹配列的值。

表单之间可以通过*外键*建立关系，列的值受这些键的约束。

8.4.1 SQL

SQL 既不是一个 API 也不是一种协议，而是一种声明式*语言*，只需要告诉它做什么即可。它是关系型数据库的通用语言。SQL 查询是客户端发送给数据库服务器的文本字符串，指明需要执行的具体操作。

SQL 语言存在很多标准定义格式，但是所有的数据库制造商都会增加它们自己的扩展，导致产生许多 SQL *方言*。如果你把数据存储在关系型数据库中，SQL 会带来一定的可移植性，但是方言和操作差异仍然会导致难以将数据移植到另一种类型的数据库中。

SQL 语句有两种主要的类型：

- DDL（*数据定义语言*）
  处理用户、数据库以及表单的创建、删除、约束和权限等。
- DML（*数据操作语言*）
  处理数据插入、选择、更新和删除。

表 8-1 列出了基本的 SQL DDL 命令。

表8-1：基本的SQL DDL命令

操作	SQL模式	SQL示例
创建数据库	`CREATE DATABASE` dbname	`CREATE DATABASE d`
选择当前数据库	`USE` dbname	`USE d`
删除数据库以及表单	`DROP DATABASE` dbname	`DROP DATABASE d`
创建表单	`CREATE TABLE` tbname (coldefs)	`CREATE TABLE t(id INT, count INT)`
删除表单	`DROP TABLE` tbname	`DROP TABLE t`
删除表单中所有的行	`TRUNCATE TABLE` tbname	`TRUNCATE TABLE t`

为什么语句中命令都是大写字母？ SQL 是不区分大小写的，但是一般为了区分命令和名称，在代码示例中还是用大写字母。

SQL 关系型数据库的主要 DML 操作可以缩略为 CRUD。

- Create：使用 `INSERT` 语句创建
- Read：使用 `SELECT` 语句选择
- Update：使用 `UPDATE` 语句更新
- Delete：使用 `DELETE` 语句删除

表 8-2 列出了一些 SQL DML 命令。

表8-2：基本的SQL DML命令

操作	SQL模式	SQL示例
增加行	`INSERT INTO` tbname `VALUES(...)`	`INSERT INTO t VALUES(7,40)`
选择全部行和全部列	`SELECT * FROM` tbname	`SELECT * FROM t`
选择全部行和部分列	`SELECT` cols `FROM` tbname	`SELECT id,count from t`
选择部分行部分列	`SELECT` cols `FROM` tbname `WHERE` condition	`SELECT id,count from t WHERE count > 5 AND id = 9`
修改一列的部分行	`UPDATE` tbname `SET` col `=` value `WHERE` condition	`UPDATE t SET count=3 WHERE id=5`
删除部分行	`DELETE FROM` tbname `WHERE` condition	`DELETE FROM t WHERE count <= 10 OR id = 16`

8.4.2 DB-API

应用程序编程接口（API）是访问某些服务的函数集合。DB-API（http://legacy.python.org/dev/peps/pep-0249/）是 Python 中访问关系型数据库的标准 API。使用它可以编写简单的程序来处理多种类型的关系型数据库，不需要为每种数据库编写独立的程序，类似于 Java 的 JDBC 或者 Perl 的 dbi。

它的主要函数如下所示。

- `connect()`

  连接数据库，包含参数用户名、密码、服务器地址，等等。

- `cursor()`

  创建一个 cursor 对象来管理查询。

- `execute()` 和 `executemany()`

  对数据库执行一个或多个 SQL 命令。

- `fetchone()`、`fetchmany()` 和 `fetchall()`

  得到 `execute` 之后的结果。

下一节的 Python 数据库模块遵循 DB-API，但会有扩展和细节上的差别。

8.4.3 SQLite

SQLite（http://www.sqlite.org）是一种轻量级的、优秀的开源关系型数据库。它是用 Python 的标准库实现，并且存储数据库在普通文件中。这些文件在不同机器和操作系统之间是可移植的，使得 SQLite 成为简易关系型数据库应用的可移植的解决方案。它不像功能全面的 MySQL 或者 PostgreSQL，SQLite 仅仅支持原生 SQL 以及多用户并发操作。浏览器、智能手机和其他应用会把 SQLite 作为嵌入数据库。

首先使用 `connect()` 函数连接本地的 SQLite 数据库文件，这个文件和目录型数据库（管理其他的表单）是等价的。字符串 `':memory:'` 仅用于在内存中创建数据库，有助于方便快速地测试，但是程序结束或者计算机关闭时所有数据都会丢失。

下一个例子会创建一个数据库 `enterprise.db` 和表单 `zoo` 用以管理路边繁华的宠物动物园业务。表单的列如下所示。

- `critter`

  可变长度的字符串，作为主键。

- `count`

  某动物的总数的整数值。

- `damages`

  人和动物的互动中损失的美元数目。

```
>>> import sqlite3
>>> conn = sqlite3.connect('enterprise.db')
```

```
>>> curs = conn.cursor()
>>> curs.execute('''CREATE TABLE zoo
    (critter VARCHAR(20) PRIMARY KEY,
     count INT,
     damages FLOAT)''')
<sqlite3.Cursor object at 0x1006a22d0>
```

Python 只有在创建长字符串时才会用到三引号（'''），例如 SQL 查询。

现在往动物园中新增一些动物：

```
>>> curs.execute('INSERT INTO zoo VALUES("duck", 5, 0.0)')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.execute('INSERT INTO zoo VALUES("bear", 2, 1000.0)')
<sqlite3.Cursor object at 0x1006a22d0>
```

使用 placeholder 是一种更安全的、插入数据的方法：

```
>>> ins = 'INSERT INTO zoo (critter, count, damages) VALUES(?, ?, ?)'
>>> curs.execute(ins, ('weasel', 1, 2000.0))
<sqlite3.Cursor object at 0x1006a22d0>
```

在 SQL 中使用三个问号表示要插入三个值，并把它们作为一个列表传入函数 execute()。这些占位符用来处理一些冗余的细节，例如引用（quoting）。它们会防止 SQL 注入：一种常见的 Web 外部攻击方式，向系统插入恶意的 SQL 命令。

现在使用 SQL 获取所有动物：

```
>>> curs.execute('SELECT * FROM zoo')
<sqlite3.Cursor object at 0x1006a22d0>
>>> rows = curs.fetchall()
>>> print(rows)
[('duck', 5, 0.0), ('bear', 2, 1000.0), ('weasel', 1, 2000.0)]
```

按照数目（count）排序，重新获得它们：

```
>>> curs.execute('SELECT * from zoo ORDER BY count')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('weasel', 1, 2000.0), ('bear', 2, 1000.0), ('duck', 5, 0.0)]
```

又需要按照降序得到它们：

```
>>> curs.execute('SELECT * from zoo ORDER BY count DESC')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('duck', 5, 0.0), ('bear', 2, 1000.0), ('weasel', 1, 2000.0)]
```

哪种类型的动物花费最多呢？

```
>>> curs.execute('''SELECT * FROM zoo WHERE
...     damages = (SELECT MAX(damages) FROM zoo)''')
<sqlite3.Cursor object at 0x1006a22d0>
>>> curs.fetchall()
[('weasel', 1, 2000.0)]
```

你可能会认为是 bears（花费最多）。实际上，最好还是查看一下实际数据。

在结束本节之前，有一点需要明确，如果我们已经打开了一个连接（connection）或者游标（cursor），不需要时应该关掉它们：

```
>>> curs.close()
>>> conn.close()
```

8.4.4 MySQL

MySQL（http://www.mysql.com）是一款非常流行的开源关系型数据库。不同于 SQLite，它是真正的数据库服务器，因此客户端可以通过网络从不同的设备连接它。

MysqlDB（http://sourceforge.net/projects/mysql-python）是最常用的 MySQL 驱动程序，但至今没有支持 Python 3。表 8-3 列出了 Python 连接 MySQL 的几个驱动程序。

表8-3：MySQL的驱动程序

名称	链接	Pypi包	导入	注意
MySQL Connector	http://dev.mysql.com/doc/connector-python/en/index.html	mysql-connector-python	`mysql.connector`	
PYMySQL	https://github.com/petehunt/PyMySQL/	pymysql	`pymysql`	
oursql	http://pythonhosted.org/oursql/	oursql	`oursql`	需要 SQL 客户端的 C 依赖库

8.4.5 PostgreSQL

PostgreSQL（http://www.postgresql.org/）是一款功能全面的开源关系型数据库，在很多方面超过 MySQL。表 8-4 列出了 Python 连接 PostgreSQL 的几个驱动程序。

表8-4：PostgreSQL的驱动程序

名称	链接	Pypi包	导入	注意
`psycopg2`	http://initd.org/psycopg/	`psycopg2`	`psycopg2`	需要来自 PostgreSQL 客户端工具的 pg_config
`py-postgresql`	http://python.projects.pgfoundry.org/	`py-postgresql`	`postgresql`	

最流行的驱动程序是 `psycopg2`，但是它的安装依赖 PostgreSQL 客户端的相关库。

8.4.6 SQLAlchemy

对于所有的关系型数据库而言，SQL 是不完全相同的，并且 DB-API 仅仅实现共有的部分。每一种数据库实现的是包含自己特征和哲学的方言。许多库函数用于消除它们之间的差异，最著名的跨数据库的 Python 库是 SQLAlchemy（http://www.sqlalchemy.org）。

它不在 Python 的标准库，但被广泛认可，使用者众多。在你的系统（Linux）中使用下面这条命令安装它：

```
$ pip install sqlalchemy
```

你可以在以下层级上使用 SQLAlchemy：

- 底层负责处理数据库连接池、执行 SQL 命令以及返回结果，这和 DB-API 相似；
- 再往上是 SQL *表达式语言*，更像 Python 的 SQL 生成器；
- 较高级的是对象关系模型（ORM），使用 SQL 表达式语言，将应用程序代码和关系型数据结构结合起来。

随着内容的深入，上面提到的术语会变得熟悉。SQLAlchemy 实现在前面几节提到的数据库驱动程序的基础上。因此不需要导入驱动程序，初始化的连接字符串会作出分配，例如：

```
dialect + driver :// user : password @ host : port / dbname
```

字符串中的值代表如下含义。

- `dialect`

  数据库类型。

- `driver`

  使用该数据库的特定驱动程序。

- `user` 和 `password`

  数据库认证字符串。

- `host` 和 `port`

  数据库服务器的位置（只有特定情况下会使用端口号 `:port`）。

- `dbname`

  初始连接到服务器中的数据库。

表 8-5 列出了常见方言和对应的驱动程序。

表8-5：SQLAlchemy连接

方言	驱动程序
sqlite	pysqlite（可以忽略）
mysql	mysqlconnector
mysql	pymysql
mysql	oursql
postgresql	psycopg2
postgresql	pypostgresql

1. 引擎层

首先，我们试用一下 SQLAlchemy 的底层，它可以实现多于基本 DB-API 的功能。

以内置于 Python 的 SQLite 为例，连接字符串忽略 `host`、`port`、`user` 和 `password`。`dbname` 表示存储 SQLite 数据库的文件，如果省去 `dbname`，SQLite 会在内存创建数据库。如果 `dbname` 以反斜线（/）开头，那么它是文件所在的绝对路径（Linux 和 OS X 是反斜线，而

在 Windows 是例如 C:\\ 的路径名）。否则它是当前目录下的相对路径。

以下是一个程序的所有部分，为了解释把它们隔开。

开始导入库函数，例子中使用了 import 的别名，用字符串 `sa` 指代 SQLAlchemy。我通常会这样做是因为 `sa` 要比 `sqlalchemy` 简洁得多：

```
>>> import sqlalchemy as sa
```

连接到数据库，并在内存中存储它（参数字符串 `'sqlite:///:memory:'` 也是可行的）：

```
>>> conn = sa.create_engine('sqlite://')
```

创建包含三列的数据库表单 `zoo`：

```
>>> conn.execute('''CREATE TABLE zoo
...     (critter VARCHAR(20) PRIMARY KEY,
...      count INT,
...      damages FLOAT)''')
<sqlalchemy.engine.result.ResultProxy object at 0x1017efb10>
```

运行函数 `conn.execute()` 返回到一个 SQLAlchemy 的对象 `ResultProxy`。马上你会看到它的用处。

顺便提一句，如果你之前从未创建过数据库表单，祝贺你，可以把它从你的人生清单（bucket list）去掉了。

现在向空表单里插入三组数据：

```
>>> ins = 'INSERT INTO zoo (critter, count, damages) VALUES (?, ?, ?)'
>>> conn.execute(ins, 'duck', 10, 0.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017efb50>
>>> conn.execute(ins, 'bear', 2, 1000.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef090>
>>> conn.execute(ins, 'weasel', 1, 2000.0)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef450>
```

接下来在数据库中查询放入的所有数据：

```
>>> rows = conn.execute('SELECT * FROM zoo')
```

在 SQLAlchemy 中，`rows` 不是一个列表，不能直接输出：

```
>>> print(rows)
<sqlalchemy.engine.result.ResultProxy object at 0x1017ef9d0>
```

但它可以像列表一样迭代，每次可以得到其中的一行：

```
>>> for row in rows:
...     print(row)
...
('duck', 10, 0.0)
('bear', 2, 1000.0)
('weasel', 1, 2000.0)
```

这个例子几乎和 SQLite DB-API 提到的示例是一样的。一个优势是在程序开始时不需要导

入数据库驱动程序，SQLAlchemy 从连接字符串（connection string）已经指定了。改变连接字符串就可以使得代码可移植到另一种数据库。另外一个优势是 SQLAlchemy 的连接池，如果想了解更多可以阅读它的文档（http://docs.sqlalchemy.org/en/latest/core/pooling.html）。

2. SQL表达式语言

再往上一层是 SQLAlchemy 的 SQL 表达式语言。它介绍了创建多种 SQL 操作的函数。相比引擎层，它能处理更多 SQL 方言的差异，对于关系型数据库应用是一种方便的中间层解决方案。

下面介绍如何创建和管理数据表 `zoo`。同样也是一个程序的连续片段。

导入和连接同之前的完全一样：

```
>>> import sqlalchemy as sa
>>> conn = sa.create_engine('sqlite://')
```

在定义表单 `zoo` 时，开始使用一些表达式语言代替 SQL：

```
>>> meta = sa.MetaData()
>>> zoo = sa.Table('zoo', meta,
...     sa.Column('critter', sa.String, primary_key=True),
...     sa.Column('count', sa.Integer),
...     sa.Column('damages', sa.Float)
...    )
>>> meta.create_all(conn)
```

注意多行调用时的圆括号。`Table()` 方法的调用结构和表单的结构相一致，此表单中包含三列，在 `Table()` 方法调用时括号内部也调用三次 `Column()`。

同时，`zoo` 是连接 SQL 数据库和 Python 数据结构的一个对象。

使用表达式语言的更多函数插入数据：

```
... conn.execute(zoo.insert(('bear', 2, 1000.0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017ea910>
>>> conn.execute(zoo.insert(('weasel', 1, 2000.0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017eab10>
>>> conn.execute(zoo.insert(('duck', 10, 0)))
<sqlalchemy.engine.result.ResultProxy object at 0x1017eac50>
```

接下来创建 SELECT 语句（`zoo.select()` 会选择出 `zoo` 对象表单的所有项，和 `SELECT * FROM zoo` 在普通 SQL 做的相同）：

```
>>> result = conn.execute(zoo.select())
```

最后得到结果：

```
>>> rows = result.fetchall()
>>> print(rows)
[('bear', 2, 1000.0), ('weasel', 1, 2000.0), ('duck', 10, 0.0)]
```

3. 对象关系映射

在上一节中，对象 `zoo` 是 SQL 和 Python 之间的中间层连接。在 SQLAlchemy 的顶层，对

象关系映射（ORM）使用 SQL 表达式语言，但尽量隐藏实际数据库的机制。你自己定义类，ORM 负责处理如何读写数据库的数据。在 ORM 这个复杂短语背后，最基本的观点是：同样使用一个关系型数据库，但操作数据的方式仍然和 Python 保持接近。

我们定义一个类 Zoo，把它挂接到 ORM。这一次，我们使用 SQLite 的 zoo.db 文件以便于验证 ORM 是否有效。

和前两节一样，代码片段实际上是一个被解释所隔开的程序。如果不明白其中的部分代码，不要着急，SQLAlchemy 文档有全部的细节，这些资料可能会变得更加复杂。这里我仅仅想让你了解使用该方法的工作量，从而方便决定本章的哪种方法更适合你。

初始的 import 还是一样，这一次需要导入新的东西：

```
>>> import sqlalchemy as sa
>>> from sqlalchemy.ext.declarative import declarative_base
```

连接数据库：

```
>>> conn = sa.create_engine('sqlite:///zoo.db')
```

现在进入 SQLAlchemy 的 ORM，定义类 Zoo，并关联它的属性和表单中的列：

```
>>> Base = declarative_base()
>>> class Zoo(Base):
...     __tablename__ = 'zoo'
...     critter = sa.Column('critter', sa.String, primary_key=True)
...     count = sa.Column('count', sa.Integer)
...     damages = sa.Column('damages', sa.Float)
...     def __init__(self, critter, count, damages):
...         self.critter = critter
...         self.count = count
...         self.damages = damages
...     def __repr__(self):
...         return "<Zoo({}, {}, {})>".format(self.critter, self.count, self.damages)
```

下面这行代码可以很神奇地创建数据库和表单：

```
>>> Base.metadata.create_all(conn)
```

然后通过创建 Python 对象插入数据，ORM 内部会管理这些：

```
>>> first = Zoo('duck', 10, 0.0)
>>> second = Zoo('bear', 2, 1000.0)
>>> third = Zoo('weasel', 1, 2000.0)
>>> first
<Zoo(duck, 10, 0.0)>
```

接下来，利用 ORM 接触 SQL，创建连接到数据库的会话（session）：

```
>>> from sqlalchemy.orm import sessionmaker
>>> Session = sessionmaker(bind=conn)
>>> session = Session()
```

借助会话，把创建的三个对象写入数据库。add() 函数增加一个对象，而 add_all() 增加一个列表：

```
>>> session.add(first)
>>> session.add_all([second, third])
```

最后使整个过程完整：

```
>>> session.commit()
```

成功了吗？好的，在当前目录下创建了文件 zoo.db，可以使用命令行的 `SQLite3` 程序验证一下：

```
$ sqlite3 zoo.db
SQLite version 3.6.12
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .tables
zoo
sqlite> select * from zoo;
duck|10|0.0
bear|2|1000.0
weasel|1|2000.0
```

本节的目的是介绍 ORM 和它在顶层的实现过程。SQLAlchemy 的作者撰写了完整的教程（http://docs.sqlalchemy.org/en/rel_0_8/orm/tutorial.html）。阅读后决定哪一级最适合你的需求：

- 普通 DB-API
- SQLAlchemy 引擎层
- SQLAlchemy 表达式语言
- SQLAlchemy ORM

使用 ORM 避开复杂的 SQL 看似是个很自然的选择。到底应该使用哪一个？有些人认为应该避免使用 ORM（http://blog.codinghorror.com/object-relational-mapping-is-the-vietnam-of-computer-science/），但其他人觉得批判太重（http://java.dzone.com/articles/martin-fowler-orm-hate）。不管谁正确，ORM 终究是一种抽象，所有的抽象在某种情况下都会出现问题，毕竟它们是有纰漏的（http://www.joelonsoftware.com/articles/LeakyAbstractions.html）。当 ORM 不能满足需求时，必须要弄明白在 SQL 如何实现修正。借用互联网的一句话：*一些人在遇到问题时理所当然地认为“我明白了，要使用 ORM”。但现在他们会有两个困扰。*谨慎使用 ORM 以及多用于简单应用，但是应用足够简单的话，或许至少可以直接使用 SQL（或者是 SQL 表达式语言）。

或者尝试一些更为简单的，例如 `dataset`（https://dataset.readthedocs.org/en/latest/）。它建立在 SQLAlchemy 之上，提供对于 SQL、JSON 以及 CSV 存储的简单 ORM。

8.5 NoSQL数据存储

有些数据库并不是关系型的，不支持 SQL。它们用来处理庞大的数据集、支持更加灵活的数据定义以及定制的数据操作。这些被统称为 NoSQL（以前的意思是 no SQL，现在理解为 not only SQL）。

8.5.1　dbm family

dbm 格式在 NoSQL 出现之前已存在很久了，它们是按照键值对的形式储存，封装在应用程序（例如网页浏览器）中，来维护各种各样的配置。从以下角度看，dbm 数据库和 Python 字典是类似的：

- 给一个键赋值，自动保存到磁盘中的数据库；
- 通过键得到对应的值。

下面简单的例子中，open() 方法的第二个参数 'r' 代表读；'w' 代表写；'c' 表示读和写，如果文件不存在则创建之：

```
>>> import dbm
>>> db = dbm.open('definitions', 'c')
```

同字典一样创建键值对，给一个键赋值：

```
>>> db['mustard'] = 'yellow'
>>> db['ketchup'] = 'red'
>>> db['pesto'] = 'green'
```

停下来看看数据库中存放了什么：

```
>>> len(db)
3
>>> db['pesto']
b'green'
```

现在关掉数据库，然后重新打开验证它是否被完整保存：

```
>>> db.close()
>>> db = dbm.open('definitions', 'r')
>>> db['mustard']
b'yellow'
```

键和值都以字节保存，因此不能对数据库对象 db 进行迭代，但是可以使用函数 len() 得到键的数目。注意 get() 和 setdefault() 函数只能用于字典的方法。

8.5.2　memcached

memcached（http://memcached.org/）是一种快速的、内存键值对象的缓存服务器。它一般置于数据库之前，用于存储网页服务器会话数据。Linux 和 OS X 点此链接（https://code.google.com/p/memcached/wiki/NewInstallFromPackage）下载，而 Windows 系统在此（http://zurmo.org/wiki/installing-memcache-on-windows）下载。如果你想要尝试使用，需要一个 memcached 服务器和 Python 的驱动程序。

当然存在很多这样的驱动程序，其中能在 Python 3 使用的是 python3-memcached（https://github.com/eguven/python3-memcached），可以通过下面这条命令安装：

```
$ pip install python-memcached
```

连接到一个 memcached 服务器之后，可以做以下事项：

- 赋值和取值
- 其中一个值的自增或者自减
- 删除其中一个键

数据在 memcached 并不是持久化保存的，后面的可能会覆盖早些写入的数据，这本来就是它的固有特性，因为它作为一个缓存服务器，通过舍弃旧数据避免程序运行时内存不足的问题。

你也可以同时连接到多个 memcached 服务器。不过下面的例子只连到一个：

```
>>> import memcache
>>> db = memcache.Client(['127.0.0.1:11211'])
>>> db.set('marco', 'polo')
True
>>> db.get('marco')
'polo'
>>> db.set('ducks', 0)
True
>>> db.get('ducks')
0
>>> db.incr('ducks', 2)
2
>>> db.get('ducks')
2
```

8.5.3 Redis

Redis（http://redis.io/）是一种数据结构服务器（data structure server）。和 memcached 类似，Redis 服务器的所有数据都是基于内存的（现在也可以选择把数据存放在磁盘）。不同于 memcached，Redis 可以实现：

- 存储数据到磁盘，方便断电重启和提升可靠性；
- 保存旧数据；
- 提供多种数据结构，不限于简单字符串。

Redis 的数据类型和 Python 很相近，Redis 服务器会是一个或多个 Python 应用程序之间共享数据的非常有帮助的中间件。据我的经验，值得用一定的篇幅介绍它。

Python 的 Redis 驱动程序 `redis-py` 在 GitHub（https://github.com/andymccurdy/redis-py）托管代码和测试用例，也可在此参考在线文档（http://redis-py.readthedocs.org/en/latest/）。可以使用这条命令安装它：

```
$ pip install redis
```

Redis 服务器自身就有好用的文档。如果在本地计算机（网络名为 `localhost`）安装和启动了 Redis 服务器，就可以开始尝试下面的程序。

1. 字符串

具有单一值的一个键被称作 Redis 的字符串。简单的 Python 数据类型可以自动转换成 Redis

字符串。现在连接到一些主机（默认 localhost）以及端口（默认 6379）上的 Redis 服务器：

```
>>> import redis
>>> conn = redis.Redis()
```

redis.Redis(‘localhost’) 或者 redis.Redis('localhost', 6379) 会得到同样的结果。

列出所有的键（目前为空）：

```
>>> conn.keys('*')
[]
```

给键 'secret' 赋值一个字符串；给键 'carats' 赋一个整数；给键 'fever' 赋一个浮点数：

```
>>> conn.set('secret', 'ni!')
True
>>> conn.set('carats', 24)
True
>>> conn.set('fever', '101.5')
True
```

通过键反过来得到对应的值：

```
>>> conn.get('secret')
b'ni!'
>>> conn.get('carats')
b'24'
>>> conn.get('fever')
b'101.5'
```

这里的 setnx() 方法只有当键不存在时才设定值：

```
>>> conn.setnx('secret', 'icky-icky-icky-ptang-zoop-boing!')
False
```

方法运行失败，因为之前已经定义了 'secret'：

```
>>> conn.get('secret')
b'ni!'
```

方法 getset() 会返回旧的值，同时赋新的值：

```
>>> conn.getset('secret', 'icky-icky-icky-ptang-zoop-boing!')
b'ni!'
```

先不急着继续下面的内容，看之前的操作是否可以运行？

```
>>> conn.get('secret')
b'icky-icky-icky-ptang-zoop-boing!'
```

使用函数 getrange() 得到子串（偏移量 offset：0 代表开始，–1 代表结束）：

```
>>> conn.getrange('secret', -6, -1)
b'boing!'
```

使用函数 setrange() 替换子串（从开始位置偏移）：

```
>>> conn.setrange('secret', 0, 'ICKY')
32
>>> conn.get('secret')
b'ICKY-icky-icky-ptang-zoop-boing!'
```

接下来使用函数 mset() 一次设置多个键值：

```
>>> conn.mset({'pie': 'cherry', 'cordial': 'sherry'})
True
```

使用函数 mget() 一次取到多个键的值：

```
>>> conn.mget(['fever', 'carats'])
[b'101.5', b'24']
```

使用函数 delete() 删掉一个键：

```
>>> conn.delete('fever')
True
```

使用函数 incr() 或者 incrbyfloat() 增加值，函数 decr() 减少值：

```
>>> conn.incr('carats')
25
>>> conn.incr('carats', 10)
35
>>> conn.decr('carats')
34
>>> conn.decr('carats', 15)
19
>>> conn.set('fever', '101.5')
True
>>> conn.incrbyfloat('fever')
102.5
>>> conn.incrbyfloat('fever', 0.5)
103.0
```

不存在函数 decrbyfloat()，可以用增加负数代替：

```
>>> conn.incrbyfloat('fever', -2.0)
101.0
```

2. 列表

Redis 的列表仅能包含字符串。当第一次插入数据时列表被创建。使用函数 lpush() 在开始处插入：

```
>>> conn.lpush('zoo', 'bear')
1
```

在开始处插入超过一项：

```
>>> conn.lpush('zoo', 'alligator', 'duck')
3
```

使用 linsert() 函数在一个值的前或者后插入：

```
>>> conn.linsert('zoo', 'before', 'bear', 'beaver')
4
>>> conn.linsert('zoo', 'after', 'bear', 'cassowary')
5
```

使用 lset() 函数在偏移量处插入（列表必须已经存在）：

```
>>> conn.lset('zoo', 2, 'marmoset')
True
```

使用 rpush() 函数在结尾处插入：

```
>>> conn.rpush('zoo', 'yak')
6
```

使用 lindex() 函数取到给定偏移量处的值：

```
>>> conn.lindex('zoo', 3)
b'bear'
```

使用 lrange() 函数取到给定偏移量范围（0~-1 代表全部）的所有值：

```
>>> conn.lrange('zoo', 0, 2)
[b'duck', b'alligator', b'marmoset']
```

使用 ltrim() 函数仅保留列表中给定范围的值：

```
>>> conn.ltrim('zoo', 1, 4)
True
```

使用函数 lrange() 得到一定范围的的值（0~-1 代表全部）：

```
>>> conn.lrange('zoo', 0, -1)
[b'alligator', b'marmoset', b'bear', b'cassowary']
```

第 10 章会介绍如何使用 Redis 列表以及*发布 – 订阅*（publish-subscribe）用于实现任务队列。

3. 哈希表

Redis 的哈希表类似于 Python 中的字典，但它仅包含字符串，因此只能有一层结构，不能进行嵌套。下面的例子创建了一个 Redis 的哈希表 song，并对它进行操作。

使用函数 hmset() 在哈希表 song 设置字段 do 和字段 re 的值：

```
>>> conn.hmset('song', {'do': 'a deer', 're': 'about a deer'})
True
```

使用函数 hset() 设置一个单一字段值：

```
>>> conn.hset('song', 'mi', 'a note to follow re')
1
```

使用函数 hget() 取到一个字段的值：

```
>>> conn.hget('song', 'mi')
b'a note to follow re'
```

使用函数 hmget() 取到多个字段的值：

```
>>> conn.hmget('song', 're', 'do')
[b'about a deer', b'a deer']
```

使用函数 hkeys() 取到所有字段的键：

```
>>> conn.hkeys('song')
[b'do', b're', b'mi']
```

使用函数 hvals() 取到所有字段的值：

```
>>> conn.hvals('song')
[b'a deer', b'about a deer', b'a note to follow re']
```

使用函数 hlen() 返回字段的总数：

```
>>> conn.hlen('song')
3
```

使用函数 hgetall() 取到所有字段的键和值：

```
>>> conn.hgetall('song')
{b'do': b'a deer', b're': b'about a deer', b'mi': b'a note to follow re'}
```

使用函数 hsetnx() 对字段中不存在的键赋值：

```
>>> conn.hsetnx('song', 'fa', 'a note that rhymes with la')
1
```

4. 集合

正如你会在下面看到的例子所示，Redis 的集合和 Python 的集合是完全类似的。

在集合中添加一个或多个值：

```
>>> conn.sadd('zoo', 'duck', 'goat', 'turkey')
3
```

取得集合中所有值的数目：

```
>>> conn.scard('zoo')
3
```

返回集合中的所有值：

```
>>> conn.smembers('zoo')
{b'duck', b'goat', b'turkey'}
```

从集合中删掉一个值：

```
>>> conn.srem('zoo', 'turkey')
True
```

新建一个集合以展示一些集合间的操作：

```
>>> conn.sadd('better_zoo', 'tiger', 'wolf', 'duck')
0
```

返回集合 zoo 和集合 better_zoo 的交集：

```
>>> conn.sinter('zoo', 'better_zoo')
{b'duck'}
```

获得集合 zoo 和集合 better_zoo 的交集，并存储到新集合 fowl_zoo：

```
>>> conn.sinterstore('fowl_zoo', 'zoo', 'better_zoo')
1
```

哪一个会在集合 fowl_zoo 中？

```
>>> conn.smembers('fowl_zoo')
{b'duck'}
```

返回集合 zoo 和集合 better_zoo 的并集：

```
>>> conn.sunion('zoo', 'better_zoo')
{b'duck', b'goat', b'wolf', b'tiger'}
```

存储并集结果到新的集合 fabulous_zoo：

```
>>> conn.sunionstore('fabulous_zoo', 'zoo', 'better_zoo')
4
>>> conn.smembers('fabulous_zoo')
{b'duck', b'goat', b'wolf', b'tiger'}
```

什么是集合 zoo 包含而集合 better_zoo 不包含的项？使用函数 sdiff() 得到它们的差集，sdiffstore() 将存储到新集合 zoo_sale：

```
>>> conn.sdiff('zoo', 'better_zoo')
{b'goat'}
>>> conn.sdiffstore('zoo_sale', 'zoo', 'better_zoo')
1
>>> conn.smembers('zoo_sale')
{b'goat'}
```

5. 有序集合

Redis 中功能最强大的数据类型之一是*有序表*（sorted set 或者 zset）。它里面的值都是独一无二的，但是每一个值都关联对应浮点值*分数*（score）。可以通过值或者分数取得每一项。有序集合有很多用途：

- 排行榜
- 二级索引
- 时间序列（把时间戳作为分数）

我们把最后一个（时间序列）作为例子，通过时间戳跟踪用户的登录。在这里，时间表达使用 Unix 的 epoch 值（更多介绍见第 10 章），它由 Python 的 time() 函数返回：

```
>>> import time
>>> now = time.time()
>>> now
1361857057.576483
```

首先增加第一个访客：

```
>>> conn.zadd('logins', 'smeagol', now)
1
```

5 分钟后，又一名访客：

```
>>> conn.zadd('logins', 'sauron', now+(5*60))
1
```

两小时后：

```
>>> conn.zadd('logins', 'bilbo', now+(2*60*60))
1
```

一天后，负载并不是很多：

```
>>> conn.zadd('logins', 'treebeard', now+(24*60*60))
1
```

那么 bilbo 登录的次序是什么？

```
>>> conn.zrank('logins', 'bilbo')
2
```

登录时间呢？

```
>>> conn.zscore('logins', 'bilbo')
1361864257.576483
```

按照登录的顺序查看每一位访客：

```
>>> conn.zrange('logins', 0, -1)
[b'smeagol', b'sauron', b'bilbo', b'treebeard']
```

附带上他们的登录时间：

```
>>> conn.zrange('logins', 0, -1, withscores=True)
[(b'smeagol', 1361857057.576483), (b'sauron', 1361857357.576483),
(b'bilbo', 1361864257.576483), (b'treebeard', 1361943457.576483)]
```

6. 位图

位图（bit）是一种非常省空间且快速的处理超大集合数字的方式。假设你有一个很多用户注册的网站，想要跟踪用户的登录频率、在某一天用户的访问量以及同一用户在固定时间内的访问频率，等等。当然，你可以使用 Redis 集合，但如果使用递增的用户 ID，位图的方法更加简洁和快速。

首先为每一天创建一个位集合（bitset）。为了测试，我们仅使用 3 天和部分用户 ID：

```
>>> days = ['2013-02-25', '2013-02-26', '2013-02-27']
>>> big_spender = 1089
>>> tire_kicker = 40459
>>> late_joiner = 550212
```

每一天是一个单独的键，对应的用户 ID 设置位，例如第一天（2013-02-25）有来自 big_spender(ID 1089) 和 tire_kicker(ID 40459) 的访问记录：

```
>>> conn.setbit(days[0], big_spender, 1)
0
>>> conn.setbit(days[0], tire_kicker, 1)
0
```

第二天用户 big_spender 又有访问：

```
>>> conn.setbit(days[1], big_spender, 1)
0
```

接下来的一天，朋友 big_spender 再次访问，并又有新人 late_joiner 访问：

```
>>> conn.setbit(days[2], big_spender, 1)
0
>>> conn.setbit(days[2], late_joiner, 1)
0
```

现在统计得到这三天的日访客数：

```
>>> for day in days:
...     conn.bitcount(day)
...
2
1
2
```

查看某一天某个用户是否有访问记录？

```
>>> conn.getbit(days[1], tire_kicker)
0
```

显然 tire_kicker 在第二天没有访问。

有多少访客每天都会访问？

```
>>> conn.bitop('and', 'everyday', *days)
68777
>>> conn.bitcount('everyday')
1
```

让你猜三次他是谁：

```
>>> conn.getbit('everyday', big_spender)
1
```

最后，这三天中独立的访客数量有多少？

```
>>> conn.bitop('or', 'alldays', *days)
68777
>>> conn.bitcount('alldays')
3
```

7. 缓存和过期

所有的 Redis 键都有一个生存期或者过期时间（expiration date），默认情况下，生存期是永久的。也可以使用 expire() 函数构造 Redis 键的生存期，下面看到的设置值是以秒为单位的数：

```
>>> import time
>>> key = 'now you see it'
>>> conn.set(key, 'but not for long')
True
>>> conn.expire(key, 5)
True
>>> conn.ttl(key)
5
>>> conn.get(key)
b'but not for long'
>>> time.sleep(6)
>>> conn.get(key)
>>>
```

expireat() 命令给一个键设定过期时间，对于更新缓存是有帮助的，并且可以限制登录会话。

8.5.4　其他的NoSQL

NoSQL 服务器都要处理远超过内存的数据，并且很多服务器要使用多台计算机。表 8-6 列出了值得注意的服务器和它们的 Python 库。

表8-6：NoSQL数据库

Site	Python API
Cassandra（http://cassandra.apache.org/）	pycassa（https://github.com/pycassa/pycassa）
CouchDB（http://couchdb.apache.org/）	couchdb-python（https://github.com/djc/couchdb-python）
HBase（http://hbase.apache.org/）	happybase（https://github.com/wbolster/happybase）
Kyoto Cabinet（http://fallabs.com/kyotocabinet/）	kyotocabinet（http://fallabs.com/kyotocabinet/pythondoc/）
MongoDB（http://www.mongodb.org/）	mongodb（http://api.mongodb.org/python/current/）
Riak（http://basho.com/riak/）	riak-python-client（https://github.com/basho/riak-python-client）

8.6　全文数据库

最后，有一类特殊的数据库用于作全文检索。它们对所有内容都建索引，所以你可以检索到吟诵风车和满车奶酪的诗歌。表 8-7 是一些流行的开源软件以及它们的 Python API。

表8-7：全文数据库

Site	Python API
Lucene（http://lucene.apache.org/）	pylucene（http://lucene.apache.org/pylucene/）
Solr（http://lucene.apache.org/solr/）	SolPython（http://wiki.apache.org/solr/SolPython）
ElasticSearch（http://www.elasticsearch.org/）	pyes（https://github.com/aparo/pyes/）
Sphinx（http://sphinxsearch.com/）	sphinxapi（https://code.google.com/p/sphinxsearch/source/browse/trunk/api/sphinxapi.py）
Xapian（http://xapian.org/）	xappy（https://code.google.com/p/xappy/）
Whoosh（https://bitbucket.org/mchaput/whoosh/wiki/Home）	由 Python 编写，包含 API

8.7 练习

(1) 将字符串 'This is a test of the emergency text system' 赋给变量 test1，然后把它写到文件 test.txt。

(2) 打开文件 test.txt，读文件内容到字符串 test2。test1 和 test2 是一样的吗？

(3) 保存这些文本到 books.csv 文件。注意，字段间是通过逗号隔开的，如果字段中含有逗号需要在整个字段加引号。

```
author,book
J R R Tolkien,The Hobbit
Lynne Truss,"Eats, Shoots & Leaves"
```

(4) 使用 csv 模块和它的 DictReader 方法读取文件 books.csv 到变量 books。输出变量 books 的值。DictReader 可以处理第二本书题目中的引号和逗号吗？

(5) 创建包含下面这些行的 CSV 文件 books.csv：

```
title,author,year
The Weirdstone of Brisingamen,Alan Garner,1960
Perdido Street Station,China Miéville,2000
Thud!,Terry Pratchett,2005
The Spellman Files,Lisa Lutz,2007
Small Gods,Terry Pratchett,1992
```

(6) 使用 sqlite3 模块创建一个 SQLite 数据库 books.db 以及包含字段 title（text）、author（text）以及 year（integer）的表单 books。

(7) 读取文件 books.csv，把数据插入到表单 book。

(8) 选择表单 book 中的 title 列，并按照字母表顺序输出。

(9) 选择表单 book 中所有的列，并按照出版顺序输出。

(10) 使用 sqlalchemy 模块连接到 sqlite3 数据库 books.db，按照 (8) 一样，选择表单 book 中的 title 列，并按照字母表顺序输出。

(11) 在你的计算机安装 Redis 服务器和 Python 的 redis 库（pip install redis）。创建一个 Redis 的哈希表 test，包含字段 count(1) 和 name('Fester Bestertester')，输出 test 的所有字段。

(12) 自增 test 的 count 字段并输出它。

第 9 章

剖析Web

CERN 是一个很适合当作 007 老巢的粒子物理研究所，横跨了法国和瑞士的边界。幸运的是，CERN 的目标并不是统治世界，而是研究宇宙的本质。这项工作给 CERN 带来了海量的数据，对物理学家和计算机科学家来说极具挑战。

1989 年，英国科学家蒂姆 · 伯纳斯 – 李首次提出可以在 CERN 和研究机构之间传递信息。他称之为*万维网*，并将其架构提炼成三个非常简单的概念。

- HTTP（*超文本传输协议*）
  规定了网络客户端和服务器[1]之间如何交换请求和响应。
- HTML（*超文本标记语言*）
  结果的展示格式。
- URL（*统一资源定位符*）
  唯一表示服务器和服务器上*资源*的方法。

在最简单的场景中，一个 Web 客户端（我觉得伯纳斯 – 李应该是第一个使用术语*浏览器*的人）通过 HTTP 连接到一个 Web 服务器，请求一个 URL，收到 HTML。

他编写了第一个 Web 浏览器和第一个 Web 服务器程序，后者部署在一台 NeXT 计算机上。NeXT 计算机是由一家短命的公司研发出来的，这家公司是乔布斯在离开苹果公司期间创办的。Web 真正开始发展壮大是在 1993 年，这一年伊利诺伊大学的一群学生发布了 Mosaic Web 浏览器（支持 Windows、Macintosh 和 Unix）和 NCSA httpd 服务器程序。当

注 1：下文会出现三个名词：服务器、Web 服务器和服务端。服务器指的是物理意义上的服务器，也就是主机或者云服务器；Web 服务器指的是服务器上负责 Web 服务的软件，比如 Nginx、Apache，等等。服务端包含服务器端的所有服务，比如 Web 服务器、API，等等。——译者注

我下载它们并用它们来搭建网站时，根本没有想到 Web 和互联网会迅速演变成我们日常生活的一部分。当时互联网仍然是官方并且非商业性的，全世界大概有 500 个公开的 Web 服务器（http://home.web.cern.ch/topics/birth-web）。到 1994 年底，Web 服务器的数量已经增长到了 10 000。互联网开始开放商业使用，Mosaic 的作者创立了 Netscape 来编写商用 Web 软件。从 Netscape 上市就可窥见当时互联网热潮的一斑，从那之后，Web 的爆发式增长就再也没有停止过。

几乎每种计算机语言都被用来编写过 Web 客户端和 Web 服务器程序。动态语言 Perl、PHP 和 Ruby 更是独领风骚。本章会说明为什么 Python 在 Web 相关的每个层面都是一种非常优秀的语言。Web 大致有三层。

- 客户端：访问远程网站。
- 服务端：为网站和 Web API 提供数据。
- Web API 和服务：用另一种不同于可视化网页的方式来交换数据。

在本章结尾的练习中，我们还会搭建一个真正的交互式网站。

9.1 Web客户端

互联网最底层的网络传输使用的是传输控制协议 / 因特网协议，更常用的叫法是 TCP/IP（详情参见 11.2.3 节）。这些协议会在计算机之间传输字节，但是并不关心这些字节的含义，后者由更高层的*协议*——用于特定目的的语法定义——来处理。HTTP 是 Web 数据交换的标准协议。

Web 是一个客户端 – 服务器系统。客户端向服务器发起*请求*：它会创建一个 TCP/IP 连接，通过 HTTP 发送 URL 和其他信息并接收一个*响应*。

响应的格式也由 HTTP 定义。其中包括请求的状态以及（如果请求成功）响应的数据和格式。

最著名的 Web 客户端是 Web *浏览器*。它可以用很多种方式来发起 HTTP 请求。你可以在地址栏中输入 URL 或者点击网页上的链接来手动发起请求。通常来说，请求所返回的数据会被当作网页展示——HTML 文档、JavaScript 文件、CSS 文件和图片——但也可以是其他类型的数据，它们并不会被用于展示。

HTTP 的一个重要概念是*无状态*。你发起的每个 HTTP 请求都和其他请求互相独立。这可以简化基本的 Web 操作，但是会让其他的操作变得更复杂。下面列举其中一些。

- *缓存*

  没有改变的远程内容应该保存在 Web 客户端中，并避免重新从服务器下载。

- Session

  购物网站必须记住你购物车中的商品。

- *认证*

  使用用户名和密码登录之后，网站应该记住你的登录状态。

可以使用 cookie 来解决无状态带来的问题。服务器可以在 cookie 中加入一些特殊的信息，

这样当客户端发回 cookie 时就可以根据 cookie 内容来进行判断。

9.1.1 使用telnet进行测试

HTTP 是基于文本的协议，所以你实际上可以自己编写协议内容来进行 Web 测试。古老的 `telnet` 程序可以让你连接到目标服务器和端口并输入命令。

下面以大家最常用的测试站点 Google 为例，来获取它的一些首页信息。输入：

```
$ telnet www.google.com 80
```

如果 google.com 的 80 端口有一个 Web 服务器程序（我觉得这是肯定的），`telnet` 会打印出一些确认信息并显示一个空白行供你输入信息：

```
Trying 74.125.225.177...
Connected to www.google.com.
Escape character is '^]'.
```

现在，向 `telnet` 中输入一条真实的 HTTP 命令并发送给 Google 的 Web 服务器。最常用的 HTTP 命令（当你在浏览器地址栏中输入 URL 时，实际上发送的就是这条命令）是 `GET`。这会获取指定资源的内容，比如一个 HTML 文件，并返回给客户端。在我们的第一次测试中，使用的是 HTTP 命令 `HEAD`，这会获取一些和资源相关的基本信息：

```
HEAD / HTTP/1.1
```

`HEAD /` 会发送 HTTP `HEAD` *动词*（命令）来获取和首页（`/`）相关的信息。再次按下回车来发送一个空行，这样远程服务器就知道你已经输入完毕，正在等待响应。你会收到一个下面这样的响应（我们用 `...` 省略了一些内容，这样页面会比较整洁）：

```
HTTP/1.1 200 OK
Date: Sat, 26 Oct 2013 17:05:17 GMT
Expires: -1
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
Set-Cookie: PREF=ID=962a70e9eb3db9d9:FF=0:TM=1382807117:LM=1382807117:S=y...
  expires=Mon, 26-Oct-2015 17:05:17 GMT;
  path=/;
  domain=.google.com
Set-Cookie: NID=67=hTvtVC7dZJmZzGktimbwVbNZxPQnaDijCz716B1L56GM9qvsqqeIGb...
  expires=Sun, 27-Apr-2014 17:05:17 GMT
  path=/;
  domain=.google.com;
  HttpOnly
P3P: CP="This is not a P3P policy! See http://www.google.com/support/accounts...
Server: gws
X-XSS-Protection: 1; mode=block
X-Frame-Options: SAMEORIGIN
Alternate-Protocol: 80:quic
Transfer-Encoding: chunked
```

这些是 HTTP 响应头和对应的值。其中一些，比如 `Date` 和 `Content-Type`，是必要的。其他的，比如 `Set-Cookie`，是用来在多次访问（稍后会讨论状态管理）之间追踪你的活动的。当

你发起 HTTP HEAD 请求时，只会得到头部。如果你使用 HTTP GET 或者 POST 命令，就会收到头部和首页的数据（混合了 HTML、CSS、JavaScript 以及其他 Google 放在首页的东西）。

我不会把你扔在 telnet 里不管，输入下面的命令来关闭 telnet：

```
q
```

9.1.2　Python的标准Web库

在 Python 2 中，Web 客户端和服务器模块结构都比较散乱。Python 3 的目标之一就是把这些模块打包成两个包（第 5 章提到过，包就是一个包含模块文件的目录）。

- http 会处理所有客户端 - 服务器 HTTP 请求的具体细节：
  - client 会处理客户端的部分
  - server 会协助你编写 Python Web 服务器程序
  - cookies 和 cookiejar 会处理 cookie，cookie 可以在请求中存储数据
- urllib 是基于 http 的高层库：
  - request 处理客户端请求
  - response 处理服务端的响应
  - parse 会解析 URL

下面，我们使用标准库来获取网站的内容。例子中的 URL 会返回一段随机文本，有点像幸运饼干[2]：

```
>>> import urllib.request as ur
>>> url = 'http://www.iheartquotes.com/api/v1/random'
>>> conn = ur.urlopen(url)
>>> print(conn)
<http.client.HTTPResponse object at 0x1006fad50>
```

在官方文档（https://docs.python.org/3/library/http.client.html）中，我们可以看到 conn 是一个包含许多方法的 HTTPResponse 对象，其中的 read() 方法会获取网页的数据：

```
>>> data = conn.read()
>>> print(data)
b'You will be surprised by a loud noise.\r\n\n[codehappy]
http://iheartquotes.com/fortune/show/20447\n'
```

这段 Python 代码会创建一个 TCP/IP 连接并连接到远程服务器，发起一个 HTTP 请求并接收 HTTP 响应。响应中除了网页的数据（那段随机文本）还包含很多其他信息，其中最重要的一条就是 HTTP 状态码：

```
>>> print(conn.status)
200
```

200 意味着一切正常。HTTP 状态码有许多种，可以根据第一个（百位）数字来分成五类。

注 2：幸运饼干，又称签语饼、幸运签语饼、幸福饼干、占卜饼等，是一种美式亚洲风味脆饼，通常由面粉、糖、香草及奶油做成，并且里面包有类似箴言或者模棱两可预言的字条。——译者注

- 1xx（信息）

  服务器收到了请求，但是需要客户端发送一些额外的信息。

- 2xx（成功）

  请求成功。除了 200 以外，其他的状态码还会包含一些特殊含义。

- 3xx（重定向）

  资源位置发生改变，所以响应会返回一个新的 URL 给客户端。

- 4xx（客户端错误）

  客户端发生错误，比如最出名的 404（页面不存在）。418（我是一个茶壶）是一个愚人节笑话。

- 5xx（服务端错误）

  500 是最常见的错误。你可能也见到过 502（网关错误），这表示 Web 服务器程序和后端的应用服务器之间无法连接。

Web 服务器程序可以返回各种格式的数据。通常是 HTML（以及一些 CSS 和 JavaScript），但是在幸运饼干例子中返回的是纯文本。数据格式由 HTTP 响应头部中的 `Content-Type` 指定，它也出现在了 google.com 例子中：

```
>>> print(conn.getheader('Content-Type'))
text/plain
```

`text/plain` 字符串表示的是一个 MIME 类型，它的意思是纯文本。google.com 例子中返回的 MIME 类型是 `text/html`。稍后你还会看到更多 MIME 类型。

出于好奇，我们来看看响应中还有什么 HTTP 头？

```
>>> for key, value in conn.getheaders():
...     print(key, value)
...
Server nginx
Date Sat, 24 Aug 2013 22:48:39 GMT
Content-Type text/plain
Transfer-Encoding chunked
Connection close
Etag "8477e32e6d053fcfdd6750f0c9c306d6"
X-Ua-Compatible IE=Edge,chrome=1
X-Runtime 0.076496
Cache-Control max-age=0, private, must-revalidate
```

还记得前面的那个 `telnet` 例子吗？现在，Python 标准库解析了 HTTP 响应的整个头部并存储成一个字典。`Date` 和 `Server` 一眼就能看懂，其他的会难懂一些。HTTP 有许多类似 `Content-Type` 的标准头部，也有许多可选头部。

9.1.3　抛开标准库：`requests`

第 1 章最开始，我们使用标准库 `urllib.request` 和 `json` 来访问 YouTube 的 API。接着使用第三方模块 `requests` 重写了一个新版本。`requests` 的版本更短并且更易读。

在大多数情况下，使用 requests 可以让 Web 客户端开发变得更加简单。你可以阅读文档（http://docs.python-requests.org/）来获取更多信息。本节会介绍 requests 的基本用法。本书之后的内容将用它来实现 Web 客户端。

首先需要把 requests 库安装到你的 Python 环境中。打开一个终端窗口（Windows 用户可以输入 cmd 来打开），输入下面的命令让 Python 的包管理器 pip 下载最新版的 requests 包并安装：

```
$ pip install requests
```

如果遇到问题，请阅读附录 D 来学习如何安装和使用 pip。

用 requests 来重写一遍上面的例子：

```
>>> import requests
>>> url = 'http://www.iheartquotes.com/api/v1/random'
>>> resp = requests.get(url)
>>> resp
<Response [200]>
>>> print(resp.text)
I know that there are people who do not love their fellow man, and I hate
people like that!

    -- Tom Lehrer, Satirist and Professor

[codehappy] http://iheartquotes.com/fortune/show/21465
```

看起来和 urllib.request.urlopen 差不多，不过我觉得看起来更简洁一些。

9.2 Web服务端

Web 开发者已经认识到，在编写 Web 服务器和服务端程序方面 Python 是一门非常优秀的语言。因此，出现了一系列基于 Python 的 Web 框架，这导致开发者很难作出选择——对 Python 教程的作者来说更是如此。

Web 框架提供了一系列用户搭建网站的特性，已经不再是一个简单的 Web（HTTP）服务器了。你会看到许多特性：路由（URL 映射到服务端函数）、模板（HTM 加上动态内容）、调试等。

我不会介绍所有的框架，只会介绍那些相对比较简单易用并且适合开发产品级网站的框架，也会介绍如何用 Python 来处理网站的动态部分并用传统的 Web 服务器来处理静态部分。

9.2.1 最简单的Python Web服务器

可以用一行 Python 代码启动一个简单的 Web 服务器：

```
$ python -m http.server
```

这是一个非常简单的 Python HTTP 服务器。如果成功启动，会打印出下面的初始化状态信息：

```
Serving HTTP on 0.0.0.0 port 8000 ...
```

0.0.0.0 表示任意 TCP 地址。这样无论服务器的地址是什么，Web 客户端都可以访问。第 11 章会有更多 TCP 的底层细节和其他的网络协议。

现在你可以通过相对路径来请求文件，它们会被 Web 服务器返回。如果你在 Web 浏览器中输入 http://localhost:8000，会看到一个目录列表，Web 服务器会打印出下面这样的访问日志：

```
127.0.0.1 - - [20/Feb/2013 22:02:37] "GET / HTTP/1.1" 200 -
```

localhost 和 127.0.0.1 在 TCP 中是同义词，都表示你的本地计算机，因此即使你的计算机没有连网，也可以执行这个例子。这行输出的具体解释如下所示。

- 127.0.0.1 是客户端的 IP 地址
- 第一个 "-" 是远程用户名，本例中为空
- 第二个 "-" 是登录用户名，本例中是可选的，为空
- [20/Feb/2013 22:02:37] 是访问日期和时间
- “GET / HTTP/1.1”是发送给 Web 服务器的命令：
    - ♦ HTTP 方法（GET）
    - ♦ 请求的资源（/，最上层目录）
    - ♦ HTTP 版本（HTTP/1.1）
- 最后的 200 表示 Web 服务器返回的 HTTP 状态码

随便点击一个文件。如果你的浏览器可以识别它的格式（HTML、PNG、GIF、JPEG 等），就会显示出这个文件，Web 服务器也会记录这次请求。举例来说，如果当前目录下有 oreilly.png 文件，请求 http://localhost:8000/oreilly.png 会返回图 7-1 中这个令人不安的家伙，Web 服务器中会显示类似下面这样的日志：

```
127.0.0.1 - - [20/Feb/2013 22:03:48] "GET /oreilly.png HTTP/1.1" 200 -
```

如果同一个目录下还有其他文件，它们也会出现在列表中，你可以点击它们来下载。如果你的浏览器可以显示点击文件的格式，那你会直接在屏幕上看到文件内容；否则你的浏览器会询问你是否要下载并保存文件。

默认的端口数是 8000，你也可以指定其他的数字：

```
$ python -m http.server 9999
```

输出如下所示：

```
Serving HTTP on 0.0.0.0 port 9999 ...
```

这个 Python 特有的 Web 服务器很适合用作快速测试。在大多数终端中，你可以按下 Ctrl+C 来结束这个进程。

一定不要把这个简单的 Web 服务器用在真正的产品级网站中。Nginx 和 Apache 等传统 Web 服务器可以更快地处理静态文件。此外，这个简单的 Web 服务器不能处理动态内容，其他更高端的 Web 服务器可以接收参数并返回动态内容。

9.2.2 Web服务器网关接口

人总是会变的，现在只提供简单的文件服务已经不能满足我们了，我们想要能够动态运行程序的 Web 服务器。在 Web 发展的早期，出现了*通用网关接口*（CGI），客户端可以通过它来让 Web 服务器运行外部程序并返回结果。CGI 也会从 Web 服务器获取用户输入的参数并传给外部程序。然而，对于每个用户请求都需要运行一次程序，这样很难扩大用户规模，因为即使程序很小，启动时还是会有明显的等待时间。

为了避免启动延迟，人们开始把语言解释器合并到 Web 服务器中。Apache 可以通过 `mod_php` 模块来运行 PHP，通过 `mod_perl` 模块来运行 Perl，通过 `mod_python` 来运行 Python。这样，动态语言的代码就可以直接在持续运行的 Apache 进程中执行，不用再调用外部程序。

另一种方式是在一个独立的持续运行的程序中运行动态语言，并让它和 Web 服务器进行通信，例如 FastCGI 和 SCGI。

Web 服务器网关接口（WSGI）的定义极大地促进了 Python 在 Web 方面的发展。WSGI 是一个通用的 API，连接 Python Web 应用和 Web 服务器。本章接下来介绍的所有 Python Web 框架和 Web 服务器都使用了 WSGI。你并不需要知道 WSGI 的原理（其实也没有多少内容），但是理解其中一些概念是非常有帮助的。

9.2.3 框架

Web 服务器会处理 HTTP 和 WSGI 的具体细节，但是真正的网站是你使用*框架*写出的 Python 代码。因此，接下来会介绍一下什么是框架，然后再来讲解如何使用它们来创建网站。

如果你想用 Python 编写网站，有许多 Python Web 框架供你选择（或许有些过多了）。对于一个 Web 框架来说，至少要具备处理客户端请求和服务端响应的能力。框架可能会具备下面这些特性中的一种或多种。

- *路由*

  解析 URL 并找到对应的服务端文件或者 Python 服务器代码。

- *模板*

  把服务端数据合并成 HTML 页面。

- *认证和授权*

  处理用户名、密码和权限。

- Session

  处理用户在多次请求之间需要存储的数据。

接下来会用两个框架（`bottle` 和 `flask`）来编写一些示例代码。之后会介绍其他框架，用它们编写带数据库的网站非常方便。无论你想编写什么网站，都能找到合适的框架。

9.2.4 Bottle

Bottle（瓶子）只包含一个简单的 Python 文件，所以非常易于使用并且易于部署。Bottle

并不是 Python 标准库的一部分，所以需要使用下面的命令来安装它：

```
$ pip install bottle
```

下面的代码会运行一个测试用于 Web 服务器，如果你在浏览器中访问 http://localhost : 9999/，这个服务器会返回一行文本。把下面的代码保存为 bottle1.py：

```
from bottle import route, run

@route('/')
def home():
  return "It isn't fancy, but it's my home page"

run(host='localhost', port=9999)
```

Bottle 使用 `route` 装饰器来关联 URL 和函数。在本例中，`/`（首页）会被 `home()` 函数处理。输入下面的命令来用 Python 运行这个服务器脚本：

```
$ python bottle1.py
```

如果你在浏览器中访问 http://lcoalhost:9999，会看到下面的内容：

```
It isn't fancy, but it's my home page
```

`run()` 函数会执行 `bottle` 内置的 Python 测试用 Web 服务器。你也可以使用其他 Web 服务器，但是在开发和测试时它非常有用。

把 HTML 硬编码到代码中是很不合适的，我们创建一个单独的 HTML 文件 index.html 并写入下面的内容：

```
My <b>new</b> and <i>improved</i> home page!!!
```

接着让 `bottle` 在首页被请求时返回这个 HTML 文件的内容。把下面的代码保存为 bottle2.py：

```
from bottle import route, run, static_file

@route('/')
def main():
    return static_file('index.html', root='.')

run(host='localhost', port=9999)
```

调用 `static_file()` 时，我们指定的是 `root` 目录（在本例中是 `'.'`，也就是当前目录）下的 index.html 文件。如果上一个例子的脚本还在执行，请终止它并运行这个新服务器：

```
$ python bottle2.py
```

在浏览器中访问 http://localhost:9999 时，会看到：

My **new** and *improved* home page!!!

再来看最后一个例子。它展示了如何指定 URL 中的参数并使用它们。按照惯例，这次的文件名是 bottle3.py：

```
from bottle import route, run, static_file

@route('/')
def home():
    return static_file('index.html', root='.')

@route('/echo/<thing>')
def echo(thing):
    return "Say hello to my little friend: %s!" % thing

run(host='localhost', port=9999)
```

我们定义了一个新函数 echo()，并且在 URL 中指定了一个字符串参数。这就是例子中 @route('/echo/<thing>') 做的事情。路由中的 <thing> 表示 URL 中 /echo/ 之后的内容都会被赋值给字符串参数 thing，然后被传入 echo 函数。下面来看看效果，如果旧服务器还在运行就终止它，然后启动新服务器：

```
$ python bottle3.py
```

接着在浏览器中访问 http://localhost:9999/echo/Mothra，会看到：

```
Say hello to my little friend: Mothra!
```

好的，现在请继续让 bottle3.py 运行，我们来做一个实验。刚才你在浏览器中验证了这个例子确实能够正常工作，并且看到了展示出来的页面。其实还可以让客户端库（比如 requests）来做同样的事。把下面的代码保存为 bottle_test.py：

```
import requests

resp = requests.get('http://localhost:9999/echo/Mothra')
if resp.status_code == 200 and \
  resp.text == 'Say hello to my little friend: Mothra!':
    print('It worked! That almost never happens!')
else:
    print('Argh, got this:', resp.text)
```

然后运行：

```
$ python bottle_test.py
```

你会在终端中看到：

```
It worked! That almost never happens!
```

这是一个简单的*单元测试*。第 8 章中详细介绍了为什么要测试以及如何用 Python 编写测试。

bottle 还有许多其他的特性，例如你可以试着在调用 run() 时加上这些参数：

- debug=True，如果出现 HTTP 错误，会创建一个调试页面；
- reloader=True，如果你修改了任何 Python 代码，浏览器中的页面会重新载入。

详细的文档可以在开发者网站（http://bottlepy.org/docs/dev/）上找到。

9.2.5 Flask

Bottle 是一个非常优秀的入门框架。但如果你需要更多的功能，那就试试 Flask 吧。Flask 最初只是 2010 年的一个愚人节玩笑，但是由于大家的反响非常热烈，作者 Armin Ronacher 把它变成了一个真正的框架。有趣的是，Flask 这个名字也是一个文字游戏[3]。

Flask 和 Bottle 一样易用，同时还支持很多专业 Web 开发需要的扩展功能，比如 Facebook 认证和数据库集成。Flask 是我最喜欢的 Python Web 框架，因为它成功地做到了既好用又强大。

Flask 包中自带了 `werkzeug` WSGI 库和 `jinja2` 模板库。你可以从终端中安装：

```
$ pip install flask
```

我们用 `flask` 来重写一下最后那个 `bottle` 例子。首先需要进行一些修改。

- Flask 默认的静态文件目录是 `static`，默认的静态文件 URL 由 `/static` 开始。我们把文件夹改成 `'.'`（当前目录），把 URL 前缀改成 `''`（空），这样 URL/ 可以被映射到文件 index.html。
- 在 `run()` 函数中，设置 `debug=True` 可以启用代码自动重载；`bottle` 把这个参数拆成了两个，`debug` 和 `reload`。

把下面的代码保存为 flask1.py：

```
from flask import Flask

app = Flask(__name__, static_folder='.', static_url_path='')

@app.route('/')
def home():
    return app.send_static_file('index.html')

@app.route('/echo/<thing>')
def echo(thing):
    return "Say hello to my little friend: %s" % thing

app.run(port=9999, debug=True)
```

然后在终端或者命令行中运行 Web 服务器：

```
$ python flask1.py
```

在浏览器中输入下面的 URL 来测试首页是否可以正常访问：

```
http://localhost:9999/
```

你能看到下面的内容（和 `bottle` 例子一样）：

My **new** and *improved* home page!!!

试试 /echo 功能：

注 3：Flask 和 bottle 都有瓶子的意思。——译者注

```
http://localhost:9999/echo/Godzilla
```

会看到：

```
Say hello to my little friend: Godzilla
```

把 debug 设置为 True 还有一个好处。在调用 run 时，如果代码中出现异常，Flask 会返回一个特殊的网页，其中会包含一些有用的信息，比如错误类型和错误位置。此外，你还可以使用一些命令来查看服务器程序中变量的值。

> 在生产环境中不要把 debug 设置为 True，否则可能会暴露出太多信息，有安全隐患。

到目前为止，Flask 的例子只是重复了我们之前用 bottle 做的事。那 Flask 能做什么 bottle 做不了的事呢？ Flask 内置了 jinja2，一个极具扩展性的模板系统。下面这个例子展示了如何在 flask 中使用 jinja2。

创建一个名为 templates 的目录，把下面的代码存为 flask2.html：

```
<html>
<head>
<title>Flask2 Example</title>
</head>
<body>
Say hello to my little friend: {{ thing }}
</body>
</html>
```

接着在服务器程序中获取这个模板，写入我们传入的值 thing，然后渲染成 HTML（为了节约空间，我去掉了 home() 函数）。把下面的代码存储为 flask2.py：

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/echo/<thing>')
def echo(thing):
    return render_template('flask2.html', thing=thing)

app.run(port=9999, debug=True)
```

thing = thing 这个参数的意思是把名为 thing 的变量传入模板，它的值是变量 thing 中的字符串。

关闭 flask1.py，运行 flask2.py：

```
$ python flask2.py
```

现在输入 URL：

```
http://localhost:9999/echo/Gamera
```

你会看到这些内容：

```
Say hello to my little friend: Gamera
```

修改一下模板内容并把它存为 flask3.html，放在 templates 目录下：

```
<html>
<head>
<title>Flask3 Example</title>
</head>
<body>
Say hello to my little friend: {{ thing }}.
Alas, it just destroyed {{ place }}!
</body>
</html>
```

你可以用很多方法把第二个参数传入 echo 的 URL。

通过URL路径传入参数

你可以把参数当作 URL 的一部分，使用这种方法可以直接扩展 URL 本身（把下面的代码存储为 flask3a.py）：

```
from flask import Flask, render_template

app = Flask(__name__)

@app.route('/echo/<thing>/<place>')
def echo(thing, place):
    return render_template('flask3.html', thing=thing, place=place)

app.run(port=9999, debug=True)
```

按照惯例，停止之前的服务器脚本并运行这个新脚本：

```
$ python flask3a.py
```

URL 看起来是这样：

```
http://localhost:9999/echo/Rodan/McKeesport
```

你会看到：

```
Say hello to my little friend: Rodan. Alas, it just destroyed McKeesport!
```

此外，还可以用 GET 参数来传递参数（把下面的代码存储为 flask3b.py）：

```
from flask import Flask, render_template, request

app = Flask(__name__)

@app.route('/echo/')
def echo():
    thing = request.args.get('thing')
    place = request.args.get('place')
    return render_template('flask3.html', thing=thing, place=place)

app.run(port=9999, debug=True)
```

运行新的服务器脚本：

```
$ python flask3b.py
```

这次使用这个 URL：

```
http://localhost:9999/echo?thing=Gorgo&place=Wilmerding
```

你会看到：

```
Say hello to my little friend: Gorgo. Alas, it just destroyed Wilmerding!
```

在 URL 中使用 GET 命令时，传入的参数形式为 &key1=val1&key2=val2&...。

你可以使用字典的 ** 操作符来向模板中一次性传入字典的多个值（把下面的代码存储为 flask3c.py）：

```
from flask import Flask, render_template,request

app = Flask(__name__)

@app.route('/echo/')
def echo():
    kwargs = {}
    kwargs['thing'] = request.args.get('thing')
    kwargs['place'] = request.args.get('place')
    return render_template('flask3.html', **kwargs)

app.run(port=9999, debug=True)
```

**kwargs 的行为与 thing=thing 和 place=place 一样，但是在参数很多时可以少输入很多内容。

jinja2 模板语言还有很多功能，如果你用过 PHP，应该会看到许多熟悉的东西。

9.2.6　非Python的Web服务器

到目前为止，我们使用的 Web 服务器都很简单：不是标准库的 http.server 就是 Bottle 和 Flask 自带的调试用服务器。在生产环境中，你需要用更快的 Web 服务器来运行 Python。下面是常用的选择：

- apache 加上 mod_wsgi 模块
- nginx 加上 uWSGI 应用服务器

两者都很不错。apache 可能是最流行的，nginx 更稳定并且占用内存更少。

1. Apache

Apache（http://httpd.apache.org/）Web 服务器中最好用的 WSGI 模块是 mod_wsgi（https://code.google.com/p/modwsgi/）。这个模块可以在 Apache 进程中运行 Python 代码，也可以在独立进程中运行 Python 代码并和 Apache 进行通信。

如果你的系统是 Linux 或者 OS X，那你已经有 apache 了。如果是 Windows，你需要安装 apache（http://httpd.apache.org/docs/current/platform/windows.html）。

最后，安装好你喜欢的基于 WSGI 的 Python Web 框架，这里我们使用 bottle。之后的工作基本上都是配置 Apache，这里有很多黑魔法。

把下面的代码存储为 /var/www/test/home.wsgi：

```
import bottle

application = bottle.default_app()

@bottle.route('/')
def home():
    return "apache and wsgi, sitting in a tree"
```

这次不要调用 run()，因为它会启动内置的 Python Web 服务器。我们需要给变量 application 赋值，因为 mod_wsgi 会使用这个变量来结合 Web 服务器和 Python 代码。

如果 apache 和 mod_wsgi 模块工作正常，只需要把它们连接到 Python 脚本就行。要做到这件事，需要向 apache 服务器的默认网站配置文件中加入一行，但是找到那个文件并不容易。它可能是 /etc/apache2/httpd.conf，也可能是 /etc/apache2/sites-available/default，还可能是某个人的宠物蝾螈的拉丁文名字。

假设现在你能找到那个文件，把下面这行加入控制默认网站的 <VirtualHost> 中：

```
WSGIScriptAlias / /var/www/test/home.wsgi
```

添加之后可能是这样的：

```
<VirtualHost *:80>
    DocumentRoot /var/www

    WSGIScriptAlias / /var/www/test/home.wsgi

    <Directory /var/www/test>
    Order allow,deny
    Allow from all
    </Directory>
</VirtualHost>
```

启动 apache，如果你已经启动就重启，这样才能应用新的配置文件。之后，如果访问 http://localhost/，你会看到：

```
apache and wsgi, sitting in a tree
```

这样就在*嵌入模式*中运行了 mod_wsgi，在这个模式下它是 apache 的一部分（在同一进程内）。

也可以用*守护模式*来运行，这样会产生一个或多个独立于 apache 的进程。要使用守护模式，可以向 apache 配置文件中加入两行内容：

```
$ WSGIDaemonProcess domain-name user=user-name group=group-name threads=25
WSGIProcessGroup domain-name
```

在上面的代码中，user-name 和 group-name 是操作系统的用户和用户组名称，domain-name

是你的互联网域名。最简单的 apache 配置如下所示：

```
<VirtualHost *:80>
    DocumentRoot /var/www

    WSGIScriptAlias / /var/www/test/home.wsgi

    WSGIDaemonProcess mydomain.com user=myuser group=mygroup threads=25
    WSGIProcessGroup mydomain.com

    <Directory /var/www/test>
    Order allow,deny
    Allow from all
    </Directory>
</VirtualHost>
```

2. Nginx Web服务器

Nginx（http://nginx.org/）Web 服务器没有内嵌的 Python 模块。它通过一个独立的 WSGI 服务器（比如 uWSGI）来和 Python 程序通信。把它们结合在一起可以实现高性能并且可配置的 Python Web 开发平台。

你可以从官网（http://wiki.nginx.org/Install）安装 nginx。此外，还需要安装 uWSGI（http://uwsgidocs.readthedocs.org/en/latest/Install.html）。uWSGI 是一个大系统，有许多需要调节的内容。可以在 http://flask.pocoo.org/docs/0.10/deploying/uwsgi/ 看到如何结合 Flask、nginx 和 uWSGI。

9.2.7 其他框架

网站和数据库就像花生酱和果冻，它们经常一起出现。小型框架，比如 bottle 和 flask，不能直接支持数据库，尽管有一些插件可以实现。

如果你需要开发基于数据库的网站并且数据库的结构不会经常变化，那最好试试大型的 Python Web 框架。现在主流的框架有以下这些。

- django（https://www.djangoproject.com/）

  是最流行的，尤其是大型网站很喜欢用它。有很多学习 django 的理由，其中最重要的就是 Python 的招聘要求中经常需要 django 的开发经验。它有 ORM 功能（8.4.6 节的“对象关系映射”部分讨论过），可以在网页中自动应用典型的数据库 CRUD 功能（创建、替换、更新和删除），就像之前在 8.4.1 节中说的一样。你也可以不用 django 自带的 ORM，可以选择 SQLAlchemy 或者直接使用 SQL 查询语句。

- web2py（http://www.web2py.com/）

  和 django 功能类似，只是风格不同。

- pyramid（http://www.pylonsproject.org/）

  诞生于最早的 pylons 项目，和 django 很像。

- turbogears（http://turbogears.org/）

  这个框架支持 ORM、多种数据库以及多种模板语言。

- `wheezy.web`（http://pythonhosted.org/wheezy.web/）
  这是一个比较新的框架，专为性能而生。在最近的测试中，它比其他框架都快（http://mindref.blogspot.com/2012/10/python-web-routing-benchmark.html）。

你可以使用这个在线表格（https://wiki.python.org/moin/WebFrameworks）来对比这些框架。

如果你的网站使用的是关系数据库，就可以不使用大型框架，直接用 `bottle`、`flask` 这类框架结合关系数据库模块就行。也可以使用 SQLAlchemy 来屏蔽数据库的差异，直接写通用 SQL 代码就行。相比特定的 ORM 语法，大多数程序员更熟悉 SQL。

当然，你完全可以不使用关系数据库。如果你的数据结构差异很大——不同行的同一列差别很大——那你或许应该试试无模式数据库，比如 8.5 节讨论过的 NoSQL 数据库。我之前开发的一个网站一开始使用 NoSQL 数据库来存储数据，后来切换到一个关系数据库，然后又切换到另一个关系数据库，接着又切换到一个 NoSQL 数据库，最后又切换回了一个关系数据库。

其他Python Web服务器

下面是一些类似 `apache` 和 `nginx` 的基于 Python 的 WSGI 服务器，使用多进程和 / 或线程（参见 11.1 节）来处理并发请求：

- `uwsgi`（http://projects.unbit.it/uwsgi/）
- `cherrypy`（http://www.cherrypy.org/）
- `pylons`（http://www.pylonsproject.org/）

下面是一些基于事件的服务器，只使用单线程但不会阻塞：

- `tornado`（http://www.tornadoweb.org）
- `gevent`（http://gevent.org/）
- `gunicorn`（http://gunicorn.org/）

第 11 章关于“并发”的讨论会详细介绍事件。

9.3 Web服务和自动化

我们已经看过了传统的 Web 客户端和服务器应用，它们会生成并使用 HTML 页面。然而，Web 逐渐演变出了许多非 HTML 的数据传输格式。

9.3.1 `webbrowser`模块

首先来看点好玩的。在终端里启动 Python 会话并输入下面的代码：

```
>>> import antigravity
```

这会调用标准库的 `webbrowser` 模块并让你的浏览器显示一个 Python 入门网页[4]。

注 4：如果你因为某些原因看不到该网页，访问 xkcd（http://xkcd.com/353/）。

你也可以直接使用这个模块。下面的程序会在浏览器中打开 Python 官网的首页：

```
>>> import webbrowser
>>> url = 'http://www.python.org/'
>>> webbrowser.open(url)
True
```

下面的代码会在新窗口中打开它：

```
>>> webbrowser.open_new(url)
True
```

如果你的浏览器支持标签，下面的代码会在新标签中打开它：

```
>>> webbrowser.open_new_tab('http://www.python.org/')
True
```

`webbrowser` 可以完全控制你的浏览器。

9.3.2　Web API和表述性状态传递

通常来说，数据只存在于网页内。如果你想获取数据，需要在 Web 浏览器中访问网页并阅读数据。如果网站作者在你最后一次访问之后做了什么改动，数据的位置和格式就可能发生变化。

除了发布网页，你还可以通过*应用编程接口*（API）来提供数据。客户端通过 URL 来访问你的服务并从响应中获取状态和数据。数据并不是 HTML 网页，而是更容易被程序处理的格式，比如 JSON 和 XML（第 8 章中对这些格式有详细的介绍）。

表述性状态传递（REST）是 Roy Fielding 在他的博士论文中提出的概念。许多产品都宣称它们具备 REST 接口或者是 RESTful 接口。在具体实现上，其实就是一个 Web 接口，即定义一组可以访问 Web 服务的 URL。

RESTful 服务会用特定的方式来使用 HTTP 动词，如下所示。

- `HEAD`

  获取资源的信息，但是不包括数据。

- `GET`

  顾名思义，`GET` 会从服务器取回资源的数据。这是浏览器使用的标准方法。如果你在 URL 中看到一个问号（?）之后跟着一堆参数，那就是一个 `GET` 请求。`GET` 不应该被用来创建、修改或者删除数据。

- `POST`

  这个动词会更新服务器上的数据。通常它会被用在 HTML 的表单和 Web API 中。

- `PUT`

  这个动词会创建一个新资源。

- DELETE

  顾名思义，这个动词会删除一些东西，就像广告中的真话一样[5]！

RESTful 客户端也可以使用 HTTP 请求头来请求一种或多种类型的内容，例如一个具备 REST 接口的复杂服务可能更希望输入和输出是 JSON 字符串。

9.3.3 JSON

第 1 章展示的两个 Python 例子获取了最热门的 YouTube 视频信息。第 8 章介绍了 JSON。JSON 非常适合用在 Web 客户端和服务器的数据交换中。它在基于 Web 的 API 中非常流行，比如 OpenStack。

9.3.4 抓取数据

有时候你可能想要更多信息——电影排名、股票价格或者商品供应信息——但是这些信息只存在于 HTML 页面中，包裹在广告和其他无关的信息中。

你可以使用下面的方法来手动提取信息：

(1) 在浏览器中输入 URL；
(2) 等待页面加载；
(3) 浏览页面并找到你想要的信息；
(4) 把信息记录下来；
(5) 可能要把这个过程重复应用在所有相关 URL 上。

然而，我们更希望能自动执行其中的一步或者多步。自动抓取 Web 信息的程序叫作*爬行者*或者*爬虫*（对于蜘蛛恐惧者来说毫无吸引力）。从远端服务器获取到信息之后爬虫会进行解析并寻找有用的信息。

如果你需要一个企业级的爬虫，那强烈推荐 Scrapy（http://scrapy.org/）：

```
$ pip install scrapy
```

Scrapy 是一个框架，并不是类似 `BeautifulSoup` 的模块。它会做更多事，不过也更难设置。更多关于 Scrapy 的信息请阅读文档（http://scrapy.org）或者在线教程（http://amaral-lab.org/blog/quick-introduction-web-crawling-using-scrapy-part-）。

9.3.5 用BeautifulSoup来抓取HTML

如果你已经拿到了一个网站的 HTML 数据并且想从中提取数据，`BeautifulSoup` 是一个不错的选择。解析 HTML 比想象中要难得多，因为互联网上的 HTML 在技术角度通常是不合法的：没有闭合的标签、不正确的嵌套以及其他复杂的东西。如果你自己尝试过用正则表达式（第 7 章介绍过）来解析 HTML，你一定遇到过这些麻烦事。

输入下面的命令来安装 `BeautifulSoup`（千万别忘了结尾的 `4`，否则 `pip` 会试着安装旧版并

注 5：意思是非常少见。——译者注

且可能会安装失败）：

```
$ pip install beautifulsoup4
```

现在可以用它来尝试一下获取一个网页上的所有链接。HTML 的 a 元素表示一个链接，它的 href 属性表示链接的目标地址。下面的例子会定义函数 get_links() 并用它来完成任务。程序可以从命令行参数接收一个或多个 URL：

```
def get_links(url):
    import requests
    from bs4 import BeautifulSoup as soup
    result = requests.get(url)
    page = result.text
    doc = soup(page)
    links = [element.get('href') for element in doc.find_all('a')]
    return links

if __name__ == '__main__':
    import sys
    for url in sys.argv[1:]:
        print('Links in', url)
        for num, link in enumerate(get_links(url), start=1):
            print(num, link)
        print()
```

我把这个程序存储为 links.py 并运行下面的命令：

```
$ python links.py http://boingboing.net
```

下面是一部分输出：

```
Links in http://boingboing.net/
1 http://boingboing.net/suggest.html
2 http://boingboing.net/category/feature/
3 http://boingboing.net/category/review/
4 http://boingboing.net/category/podcasts
5 http://boingboing.net/category/video/
6 http://bbs.boingboing.net/
7 javascript:void(0)
8 http://shop.boingboing.net/
9 http://boingboing.net/about
10 http://boingboing.net/contact
```

9.4　练习

(1) 如果你还没有安装 flask，现在安装它。这样会自动安装 werkzeug、jinja2 和其他包。

(2) 搭建一个网站框架，使用 Flask 的调试 / 代码重载来开发 Web 服务器。使用主机名 localhost 和默认端口 5000 来启动服务器。如果你电脑的 5000 端口已经被占用，使用其他端口。

(3) 添加一个 home() 函数来处理对于主页的请求，让它返回字符串 It's alive!。

(4) 创建一个名为 home.html 的 Jinja2 模板文件，内容如下所示：

```
<html>
<head>
<title>It's alive!</title>
<body>
I'm of course referring to {{thing}}, which is {{height}} feet tall and {{color}}.
</body>
</html>
```

(5) 修改 `home()` 函数，让它使用 home.html 模板。给模板传入三个 `GET` 参数：`thing`、`height` 和 `color`。

第10章

系统

“被密封在一个仓库的纸箱中是一件计算机能做但是大多数人无法做到的事。”

——Jack Handey

在你日常使用计算机时，经常需要列出一个文件夹或者目录的内容，创建和删除文件，以及做其他一些比较无聊但是不得不做的“家务活”。在 Python 程序中可以做到同样的事，甚至能做更多的事。这些功能是否能减少你的工作量呢？我们拭目以待。

Python 在模块 `os`（操作系统，operating system）中提供了许多系统函数，本章的所有程序都需要导入这个模块。

10.1　文件

和其他语言一样，Python 的文件操作很像 Unix。有些函数的名字相同，比如 `chown()` 和 `chmod()`，不过也有很多新函数。

10.1.1　用open()创建文件

8.1 节介绍了如何使用 `open()` 函数来打开文件或者创建文件。下面来创建一个名为 oops.txt 的文本文件：

```
>>> fout = open('oops.txt', 'wt')
>>> print('Oops, I created a file.', file=fout)
>>> fout.close()
```

我们用这个文件来进行一些测试。

10.1.2 用exists()检查文件是否存在

要判断文件或者目录是否存在，可以使用 exists()，传入相对或者绝对路径名，如下所示：

```
>>> import os
>>> os.path.exists('oops.txt')
True
>>> os.path.exists('./oops.txt')
True
>>> os.path.exists('waffles')
False
>>> os.path.exists('.')
True
>>> os.path.exists('..')
True
```

10.1.3 用isfile()检查是否为文件

本节中的函数可以检查一个名称是文件、目录还是符号链接（详情见下方的例子）。

我们要学习的第一个函数是 isfile，它只回答一个问题：这个是不是文件？

```
>>> name = 'oops.txt'
>>> os.path.isfile(name)
True
```

下面是判断目录的方法：

```
>>> os.path.isdir(name)
False
```

一个点号（.）表示当前目录，两个点号（..）表示上层目录。它们一直存在，所以下面的语句总会返回 True：

```
>>> os.path.isdir('.')
True
```

os 模块中有许多处理路径名（完整的文件名，由 / 开始并包含所有上级目录）的函数。其中之一是 isabs()，可以判断参数是否是一个绝对路径名。参数不需要是一个真正的文件：

```
>>> os.path.isabs(name)
False
>>> os.path.isabs('/big/fake/name')
True
>>> os.path.isabs('big/fake/name/without/a/leading/slash')
False
```

10.1.4 用copy()复制文件

copy() 函数来自于另一个模块 shutil。下面的例子会把文件 oops.txt 复制到文件 ohno.txt：

```
>>> import shutil
>>> shutil.copy('oops.txt', 'ohno.txt')
```

shutil.move() 函数会复制一个文件并删除原始文件。

10.1.5　用rename()重命名文件

这个函数的作用看名字就知道了。下面的例子会把 ohno.txt 文件重命名为 ohwell.txt 文件：

```
>>> import os
>>> os.rename('ohno.txt', 'ohwell.txt')
```

10.1.6　用link()或者symlink()创建链接

在 Unix 中，文件只存在于一个位置，但是可以有多个名称，这种机制叫作*链接*。在更底层的*硬链接*中，找到一个文件的所有名称并不容易。可以考虑使用*符号链接*，它会把新名称当作文件存储，这样一次就可以获取到原始名称和新名称。link() 会创建一个硬链接。symlink() 会创建一个符号链接。islink() 函数会检查参数是文件还是符号链接。

下面这个例子展示了如何把已有文件 oops.txt 硬链接到一个新文件 yikes.txt：

```
>>> os.link('oops.txt', 'yikes.txt')
>>> os.path.isfile('yikes.txt')
True
```

下面的例子展示了如何把已有文件 oops.txt 符号链接到一个新文件 jeepers.txt：

```
>>> os.path.islink('yikes.txt')
False
>>> os.symlink('oops.txt', 'jeepers.txt')
>>> os.path.islink('jeepers.txt')
True
```

10.1.7　用chmod()修改权限

在 Unix 系统中，chmod() 可以修改文件权限。可以设置用户（如果是你创建了文件，那用户通常就是你）、当前用户所在用户组以及其他所有用户组的读、写和执行权限。这个命令接收一个压缩过的八进制（基数为 8）值，这个值包含用户、用户组和权限。举例来说，下面的命令可以让 oops.txt 只能被拥有者读：

```
>>> os.chmod('oops.txt', 0o400)
```

如果你不想用压缩过的八进制值并且更喜欢那些（有点）难懂的符号，可以从 stat 模块中导入一些常量并用在语句中：

```
>>> import stat
>>> os.chmod('oops.txt', stat.S_IRUSR)
```

10.1.8　用chown()修改所有者

这个函数也是 Unix/Linux/Mac 特有的。你可以指定用户的 ID（uid）和用户组 ID（gid）来修改文件的所有者和 / 或所有用户组：

```
>>> uid = 5
>>> gid = 22
>>> os.chown('oops', uid, gid)
```

10.1.9　用abspath()获取路径名

这个函数会把一个相对路径名扩展成绝对路径名。如果你的当前目录是 /usr/gabenlunzie，其中有文件 oops.txt，那你可以用这个函数得到下面的输出：

```
>>> os.path.abspath('oops.txt')
'/usr/gaberlunzie/oops.txt'
```

10.1.10　用realpath()获取符号的路径名

在之前的几节中，我们创建了一个 oops.txt 到 jeepers.txt 的符号链接。在这种情况下，你可以使用 `realpath()` 函数从 jeepers.txt 获取名称 oops.txt，如下所示：

```
>>> os.path.realpath('jeepers.txt')
'/usr/gaberlunzie/oops.txt'
```

10.1.11　用remove()删除文件

下面的例子使用 `remove()` 函数来删除 oops.txt 文件：

```
>>> os.remove('oops.txt')
>>> os.path.exists('oops.txt')
False
```

10.2　目录

在大多数操作系统中，文件被存储在多级目录（现在经常被称为*文件夹*）中。包含所有这些文件和目录的容器是*文件系统*（有时候被称为*卷*）。标准模块 `os` 可以处理这些东西，下面是一些可以使用的函数。

10.2.1　使用mkdir()创建目录

下面的例子展示了如何创建目录 `poems`：

```
>>> os.mkdir('poems')
>>> os.path.exists('poems')
True
```

10.2.2　使用rmdir()删除目录

假如你现在发现不需要这个目录，可以用这个函数来删除目录：

```
>>> os.rmdir('poems')
>>> os.path.exists('poems')
False
```

10.2.3　使用listdir()列出目录内容

你又后悔了，来重新创建 poems 并加入一些内容：

```
>>> os.mkdir('poems')
```

现在列出它的内容（目前还是空）：

```
>>> os.listdir('poems')
[]
```

接着创建一个子目录：

```
>>> os.mkdir('poems/mcintyre')
>>> os.listdir('poems')
['mcintyre']
```

在这个子目录中，创建一个文件（不要手动输入这些文字，除非你真的很喜欢这首诗；一定要确保文中的单引号和三引号可以正确匹配）：

```
>>> fout = open('poems/mcintyre/the_good_man', 'wt')
>>> fout.write('''Cheerful and happy was his mood,
... He to the poor was kind and good,
... And he oft' times did find them food,
... Also supplies of coal and wood,
... He never spake a word was rude,
... And cheer'd those did o'er sorrows brood,
... He passed away not understood,
... Because no poet in his lays
... Had penned a sonnet in his praise,
... 'Tis sad, but such is world's ways.
... ''')
344
>>> fout.close()
```

最后来看看目录中有什么：

```
>>> os.listdir('poems/mcintyre')
['the_good_man']
```

10.2.4　使用chdir()修改当前目录

你可以使用这个函数从一个目录跳转到另一个目录。我们从当前目录跳转到 poems 目录中：

```
>>> import os
>>> os.chdir('poems')
>>> os.listdir('.')
['mcintyre']
```

10.2.5　使用glob()列出匹配文件

glob() 函数会使用 Unix shell 的规则来匹配文件或者目录，而不是更复杂的正则表达式。具体规则如下所示：

- * 会匹配任意名称（re 中是 .*）
- ? 会匹配一个字符
- [abc] 会匹配字符 a、b 和 c
- [!abc] 会匹配除了 a、b 和 c 之外的所有字符

试着获取所有以 m 开头的文件和目录：

```
>>> import glob
>>> glob.glob('m*')
['mcintyre']
```

获取所有名称为两个字符的文件和目录：

```
>>> glob.glob('??')
[]
```

获取名称为 8 个字符并且以 m 开头和以 e 结尾的文件和目录：

```
>>> glob.glob('m??????e')
['mcintyre']
```

获取所有以 k、l 或者 m 开头并且以 e 结尾的文件和目录：

```
>>> glob.glob('[klm]*e')
['mcintyre']
```

10.3 程序和进程

当运行一个程序时，操作系统会创建一个进程。它会使用系统资源（CPU、内存和磁盘空间）和操作系统内核中的数据结构（文件、网络连接、用量统计等）。进程之间是互相隔离的，即一个进程既无法访问其他进程的内容，也无法操作其他进程。

操作系统会跟踪所有正在运行的进程，给每个进程一小段运行时间，然后切换到其他进程，这样既可以做到公平又可以响应用户操作。你可以在图形界面中查看进程状态，在 Mac OS X 上可以使用活动监视器，在 Windows 上可以使用任务管理器。

你也可以自己编写程序来获取进程信息。标准库模块 os 提供了一些常用的获取系统信息的函数。举例来说，下面的函数会获取正在运行的 Python 解释器的进程号和当前工作目录：

```
>>> import os
>>> os.getpid()
76051
>>> os.getcwd()
'/Users/williamlubanovic'
```

下面的函数会获取我的用户 ID 和用户组 ID：

```
>>> os.getuid()
501
>>> os.getgid()
20
```

10.3.1　使用subprocess创建进程

到目前为止，你看到的所有程序都是单进程程序。你可以使用 Python 标准库中的 subprocess 模块来启动和终止其他程序。如果只是想在 shell 中运行其他程序并获取它的输出（标准输出和标准错误输出），可以使用 getoutput() 函数。这里获取了 Unix date 程序的输出：

```
>>> import subprocess
>>> ret = subprocess.getoutput('date')
>>> ret
'Sun Mar 30 22:54:37 CDT 2014'
```

在进程执行完毕之前，你获取不到任何内容。如果需要调用一些比较耗时的程序，可以使用 11.1 节提到的并发。因为 getoutput() 的参数是一个字符串，可以表示一个完整的 shell 命令，所以你可以在里面使用参数、管道、I/O 重定向 < 和 >，等等：

```
>>> ret = subprocess.getoutput('date -u')
>>> ret
'Mon Mar 31 03:55:01 UTC 2014'
```

把这个输出用管道传给 wc 命令，可以计算出一共有 1 行、6 个单词和 29 个字符：

```
>>> ret = subprocess.getoutput('date -u | wc')
>>> ret
'       1       6      29'
```

另一个类似的方法是 check_output()，可以接受一个命令和参数列表。默认情况下，它返回的不是字符串，而是字节类型的标准输出。此外，这个函数并没有使用 shell：

```
>>> ret = subprocess.check_output(['date', '-u'])
>>> ret
b'Mon Mar 31 04:01:50 UTC 2014\n'
```

如果要获取其他程序的退出状态，可以使用 getstatusoutput() 函数，它会返回一个包含状态码和输出的元组：

```
>>> ret = subprocess.getstatusoutput('date')
>>> ret
(0, 'Sat Jan 18 21:36:23 CST 2014')
```

如果只想要退出状态，可以使用 call()：

```
>>> ret = subprocess.call('date')
Sat Jan 18 21:33:11 CST 2014
>>> ret
0
```

（在 Unix 类操作系统中，退出状态 0 通常表示运行成功。）

本例中，日期和时间被打印到输出中，但是并没有被我们的程序获取，所以 ret 中只有状态码。

你可以用两种方式来运行带参数的程序。第一种是在字符串中写明参数。我们的示例中使

用的是 date -u，这会打印出 UTC 格式的当前日期和时间（稍后会有 UTC 的解释）：

```
>>> ret = subprocess.call('date -u', shell=True)
Tue Jan 21 04:40:04 UTC 2014
```

需要加上参数 shell=True，这样函数就会用 shell 来执行命令，shell 会把 date -u 分割成单独的字符串并对通配符（比如 *，我们的例子中没有使用它）进行扩展。

第二种方式是传入一个参数列表，这样函数就不需要调用 shell：

```
>>> ret = subprocess.call(['date', '-u'])
Tue Jan 21 04:41:59 UTC 2014
```

10.3.2 使用multiprocessing创建进程

你可以在一个单独的进程中运行一个 Python 函数，也可以使用 multiprocessing 模块在一个程序中运行多个进程。下面的例子做了一些很无聊的事，把它存储为 mp.py，然后用 python mp.py 运行它：

```
import multiprocessing
import os

def do_this(what):
    whoami(what)

def whoami(what):
    print("Process %s says: %s" % (os.getpid(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = multiprocessing.Process(target=do_this,
          args=("I'm function %s" % n,))
        p.start()
```

运行时得到如下输出：

```
Process 6224 says: I'm the main program
Process 6225 says: I'm function 0
Process 6226 says: I'm function 1
Process 6227 says: I'm function 2
Process 6228 says: I'm function 3
```

Process() 函数会创建一个新进程来运行 do_this() 函数。由于我们在一个循环中执行它，所以生成了 4 个执行 do_this() 的进程并在执行完毕后退出。

multiprocessing 模块真正的功能比这个例子中所展示的要强大得多。当你需要用多进程来减少运行时间时，它非常有用，比如下载需要抓取的网页、调整图片尺寸等。它支持任务队列和进程间通信，而且可以等待所有进程执行完毕。11.1 节会详细讲解。

10.3.3 使用terminate()终止进程

如果创建了一个或者多个进程并且想终止它们（可能是因为进入了死循环，也可能是因

为你觉得很无聊，还有可能是因为你只是想干坏事），可以使用 terminate()。下面的例子中，我们的进程会一直计数到百万，每次计数之后都会等待 1 秒并打印出相关信息。然而，主程序只会保持 5 秒耐心，之后就会终止进程：

```
import multiprocessing
import time
import os

def whoami(name):
    print("I'm %s, in process %s" % (name, os.getpid()))

def loopy(name):
    whoami(name)
    start = 1
    stop = 1000000
    for num in range(start, stop):
        print("\tNumber %s of %s. Honk!" % (num, stop))
        time.sleep(1)

if __name__ == "__main__":
    whoami("main")
    p = multiprocessing.Process(target=loopy, args=("loopy",))
    p.start()
    time.sleep(5)
    p.terminate()
```

运行程序时得到如下输出：

```
I'm main, in process 97080
I'm loopy, in process 97081
    Number 1 of 1000000. Honk!
    Number 2 of 1000000. Honk!
    Number 3 of 1000000. Honk!
    Number 4 of 1000000. Honk!
    Number 5 of 1000000. Honk!
```

10.4 日期和时间

程序员们需要花费大量时间来处理日期和时间。我们会讨论一些常见的问题，之后会介绍一些对应的最佳实践和能够帮助缓解问题的小技巧。

可以用多种方式来表示日期，甚至多到让人厌烦。即使是使用罗马历的英语国家也有很多表示日期的方法：

- July 29 1984
- 29 Jul 1984
- 29/7/1984
- 7/29/1984

表示日期的第一个问题就是二义性。从上面的例子可以很容易看出，7 表示月份，29 表示天数，因为月份不可能是 29。但是 1/6/2012 呢？它到底是 1 月 6 日还是 6 月 1 日呢？

罗马历中，月份的名字在不同语言中差别很大。在其他文化中，年份和月份本身的定义都有可能不一样。

闰年是另一个问题。你可能知道，每隔 4 年会出现一个闰年（夏季奥运会和美国总统选举也是隔四年一次）。不过你知道吗，年份是整百数时，必须是 400 的倍数才是闰年。下面的代码可以检测是否是闰年：

```
>>> import calendar
>>> calendar.isleap(1900)
False
>>> calendar.isleap(1996)
True
>>> calendar.isleap(1999)
False
>>> calendar.isleap(2000)
True
>>> calendar.isleap(2002)
False
>>> calendar.isleap(2004)
True
```

时间则有另外的问题，主要是因为时区以及夏时制。观察时区图会发现，时区是按照政治和历史因素分割的，并不是按照经度 15 度（360 度 /24）分割。不同国家每年夏时制的开始和结束时间也不一样。实际上，南半球和北半球的夏时制时间段刚好相反（细想就知道原因了）。

Python 的标准库中有很多日期和时间模块：datetime、time、calendar、dateutil，等等。有些在功能上有重复，还有点不好理解。

10.4.1　datetime模块

首先介绍一下标准 datetime 模块。它定义了 4 个主要的对象，每个对象都有很多方法：

- date 处理年、月、日
- time 处理时、分、秒和分数
- datetime 处理日期和时间同时出现的情况
- timedelta 处理日期和 / 或时间间隔

你可以指定年、月、日，来创建一个 date 对象。这些值之后会变成对象的属性：

```
>>> from datetime import date
>>> halloween = date(2014, 10, 31)
>>> halloween
datetime.date(2014, 10, 31)
>>> halloween.day
31
>>> halloween.month
10
>>> halloween.year
2014
```

可以使用 isoformat() 方法打印一个 date 对象：

```
>>> halloween.isoformat()
'2014-10-31'
```

iso 是指 ISO 8601，一种表示日期和时间的国际标准。这个标准的显示顺序是从一般（年）到特殊（日）。它也可以对日期进行正确的排序：先按照年，然后是月，最后是日。我经常在程序中使用这种格式表示日期，还会在存储和日期相关的数据时当作文件名。下一节会介绍更复杂的 strptime() 和 strftime() 方法，它们分别用来解析和格式化日期。

下面的例子使用 today() 方法生成今天的日期：

```
>>> from datetime import date
>>> now = date.today()
>>> now
datetime.date(2014, 2, 2)
```

下面的例子使用 timedelta 对象来实现 date 的加法：

```
>>> from datetime import timedelta
>>> one_day = timedelta(days=1)
>>> tomorrow = now + one_day
>>> tomorrow
datetime.date(2014, 2, 3)
>>> now + 17*one_day
datetime.date(2014, 2, 19)
>>> yesterday = now - one_day
>>> yesterday
datetime.date(2014, 2, 1)
```

date 的范围是 date.min（年 = 1，月 = 1，日 = 1）到 date.max（年 = 9999，月 = 12，日 = 31）。因此，不能使用它来进行和历史或者天文相关的计算。

datetime 模块中的 time 对象用来表示一天中的时间：

```
>>> from datetime import time
>>> noon = time(12, 0, 0)
>>> noon
datetime.time(12, 0)
>>> noon.hour
12
>>> noon.minute
0
>>> noon.second
0
>>> noon.microsecond
0
```

参数的顺序是按照时间单位从大（时）到小（微秒）排列。如果没有参数，time 会默认全部使用 0。顺便说一句，能够存取微秒并不意味着你能从计算机中得到准确的微秒。每秒的准确度取决于硬件和操作系统中的很多因素。

datetime 对象既包含日期也包含时间。你可以直接创建一个 datetime 对象，如下所示，表示 2014 年 1 月 2 日上午 3:04，5 秒 6 微秒：

```
>>> from datetime import datetime
>>> some_day = datetime(2014, 1, 2, 3, 4, 5, 6)
>>> some_day
datetime.datetime(2014, 1, 2, 3, 4, 5, 6)
```

datetime 对象也有一个 isoformat() 方法：

```
>>> some_day.isoformat()
'2014-01-02T03:04:05.000006'
```

中间的 T 把日期和时间分割开。

datetime 有一个 now() 方法，可以用它获取当前日期和时间：

```
>>> from datetime import datetime
>>> now = datetime.now()
>>> now
datetime.datetime(2014, 2, 2, 23, 15, 34, 694988)
14
>>> now.month
2
>>> now.day
2
>>> now.hour
23
>>> now.minute
15
>>> now.second
34
>>> now.microsecond
694988
```

可以使用 combine() 方法把一个 date 对象和一个 time 对象合并成一个 datetime 对象：

```
>>> from datetime import datetime, time, date
>>> noon = time(12)
>>> this_day = date.today()
>>> noon_today = datetime.combine(this_day, noon)
>>> noon_today
datetime.datetime(2014, 2, 2, 12, 0)
```

也可以使用 date() 和 time() 方法从 datetime 中取出 date 和 time 部分：

```
>>> noon_today.date()
datetime.date(2014, 2, 2)
>>> noon_today.time()
datetime.time(12, 0)
```

10.4.2 使用time模块

Python 的 datetime 模块中有一个 time 对象，Python 还有一个单独的 time 模块，这有点令人困扰。此外，time 模块还有一个函数叫作 time()。

一种表示绝对时间的方法是计算从某个起始点开始的秒数。Unix 时间使用的是从 1970 年 1 月 1 日 0 点开始的秒数[1]。这个值通常被称为*纪元*，它是不同系统之间最简单的交换日期时间的方法。

time 模块的 time() 函数会返回当前时间的纪元值：

```
>>> import time
>>> now = time.time()
>>> now
1391488263.664645
```

数一下位数，从 1970 年新年[2]到现在已经过去了超过十亿秒。时间都去哪儿了？

可以用 ctime() 把一个纪元值转换成一个字符串：

```
>>> time.ctime(now)
'Mon Feb  3 22:31:03 2014'
```

下一节会介绍如何生成其他格式的日期和时间。

纪元值是不同系统（比如 JavaScript）之间交换日期和时间的一种非常有用的方法。但是，有时候想要真正的日期而不是一串数字，这时可以使用 struct_time 对象。localtime() 会返回当前系统时区下的时间，gmtime() 会返回 UTC 时间：

```
>>> time.localtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=3, tm_hour=22, tm_min=31,
tm_sec=3, tm_wday=0, tm_yday=34, tm_isdst=0)
>>> time.gmtime(now)
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=4, tm_hour=4, tm_min=31,
tm_sec=3, tm_wday=1, tm_yday=35, tm_isdst=0)
```

在我所处的时区（中央时区），22:31 是 UTC（正式称呼是*格林威治时间*或者*祖鲁时间*）第二天的 04:31。如果省略 localtime() 或者 gmtime() 的参数，默认会返回当前时间。

对应上面两个函数的是 mktime()，它会把 struct_time 对象转换回纪元值：

```
>>> tm = time.localtime(now)
>>> time.mktime(tm)
1391488263.0
```

这个值和之前 now() 返回的纪元值并不完全相同，因为 struct_time 对象只能精确到秒。

一些建议：尽量多使用 UTC 来代替时区。UTC 是绝对时间，和时区无关。如果你有服务器，把它的时间设置为 UTC，不要使用本地时间。

还有一些建议（都是免费的）：如果有可能，*绝对不使用夏时制时间*。如果使用了夏时制时间，那每年的一段时间（春季）会少一个小时，另一段时间（秋季）会多一个小时。出于某些原因，许多组织在它们的计算机系统中使用夏时制，但是每年都需要处理数据重复和丢失。说起来都是泪。

注 1：开始时间大约与 Unix 出现的时间吻合。

注 2：美国新年是中国的元旦。——译者注

别忘了，UTC 指时间，UTF-8 指字符串（更多关于 UTF-8 的内容参见第 7 章）。

10.4.3 读写日期和时间

`isoformat()` 并不是唯一一种可以打印日期和时间的方法。前面已经见过，`time` 模块中的 `ctime()` 函数可以把纪元转换成字符串：

```
>>> import time
>>> now = time.time()
>>> time.ctime(now)
'Mon Feb  3 21:14:36 2014'
```

也可以使用 `strftime()` 把日期和时间转换成字符串。这个方法在 `datetime`、`date` 和 `time` 对象中都有，在 `time` 模块中也有。`strftime()` 使用格式化字符串来指定输出，详见表 10-1。

表10-1：`strftime()`的格式化字符串

格式化字符串	日期/时间单元	范围
%Y	年	1900-...
%m	月	01-12
%B	月名	January,...
%b	月名缩写	Jan,...
%d	日	01-31
%A	星期	Sunday,...
%a	星期缩写	Sun,...
%H	时（24 小时制）	00-23
%I	时（12 小时制）	01-12
%p	上午 / 下午	AM, PM
%M	分	00-59
%S	秒	00-59

数字都是左侧补零。

下面是 `time` 模块中的 `strftime()` 函数。它会把 `struct_time` 对象转换成字符串。我们首先定义格式化字符串 `fmt`，然后使用它：

```
>>> import time
>>> fmt = "It's %A, %B %d, %Y, local time %I:%M:%S%p"
>>> t = time.localtime()
>>> t
time.struct_time(tm_year=2014, tm_mon=2, tm_mday=4, tm_hour=19,
tm_min=28, tm_sec=38, tm_wday=1, tm_yday=35, tm_isdst=0)
>>> time.strftime(fmt, t)
"It's Tuesday, February 04, 2014, local time 07:28:38PM"
```

如果使用 `date` 对象的 `strftime()` 函数，只能获取日期部分，时间默认是午夜：

```
>>> from datetime import date
>>> some_day = date(2014, 7, 4)
>>> fmt = "It's %B %d, %Y, local time %I:%M:%S%p"
>>> some_day.strftime(fmt)
"It's Friday, July 04, 2014, local time 12:00:00AM"
```

对于 time 对象，只会转换时间部分：

```
>>> from datetime import time
>>> some_time = time(10, 35)
>>> some_time.strftime(fmt)
"It's Monday, January 01, 1900, local time 10:35:00AM"
```

显然，你肯定不想使用 time 对象的时间部分，因为它们毫无意义。

如果要把字符串转换成日期或者时间，可以对同样的格式化字符串使用 strptime() 函数。这里不能使用正则表达式，字符串的非格式化部分（没有 % 的部分）必须完全匹配。假设可以匹配格式“年 – 月 – 日”，比如 2012-01-29，如果目标字符串中用空格代替破折号，解析时会发生什么呢？

```
>>> import time
>>> fmt = "%Y-%m-%d"
>>> time.strptime("2012 01 29", fmt)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/
  python3.3/_strptime.py", line 494, in _strptime_time
    tt = _strptime(data_string, format)[0]
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/
  python3.3/_strptime.py", line 337, in _strptime
    (data_string, format))
ValueError: time data '2012 01 29' does not match format '%Y-%m-%d'
```

如果传入 strptime() 的字符串中有破折号会怎么样呢？

```
>>> time.strptime("2012-01-29", fmt)
time.struct_time(tm_year=2012, tm_mon=1, tm_mday=29, tm_hour=0, tm_min=0,
tm_sec=0, tm_wday=6, tm_yday=29, tm_isdst=-1)
```

现在可以正常解析了。

即使字符串看起来可以匹配格式，但如果超出取值范围也会报错：

```
>>> time.strptime("2012-13-29", fmt)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/
  python3.3/_strptime.py", line 494, in _strptime_time
    tt = _strptime(data_string, format)[0]
  File "/Library/Frameworks/Python.framework/Versions/3.3/lib/
  python3.3/_strptime.py", line 337, in _strptime
    (data_string, format))
ValueError: time data '2012-13-29' does not match format '%Y-%m-%d'
```

名称通过你的 locale 设置，这是操作系统中的国际化设置。如果要打印出不同的月和日名

称，可以通过 setlocale() 来修改。这个函数的第一个参数是 locale.LC_TIME，表示设置的是日期和时间，第二个参数是一个结合了语言和国家名称的缩写字符串。假设我们邀请了一些国际友人来参加万圣节派对，需要分别打印出美国英语、法语、德语、西班牙语和冰岛语中的月、日和具体的星期。（什么？你觉得冰岛人并不喜欢这样的派对？错！他们甚至有真正的精灵。）

```
>>> import locale
>>> from datetime import date
>>> halloween = date(2014, 10, 31)
>>> for lang_country in ['en_us', 'fr_fr', 'de_de', 'es_es', 'is_is',]:
...     locale.setlocale(locale.LC_TIME, lang_country)
...     halloween.strftime('%A, %B %d')
...
'en_us'
'Friday, October 31'
'fr_fr'
'Vendredi, octobre 31'
'de_de'
'Freitag, Oktober 31'
'es_es'
'viernes, octubre 31'
'is_is'
'föstudagur, október 31'
>>>
```

去哪儿找这些神奇的 lang_country 值呢？虽然不是特别靠谱，不过可以用下面的方式来获取所有值（有好几百种）：

```
>>> import locale
>>> names = locale.locale_alias.keys()
```

我们从 names 中过滤出可以用在 setlocale() 中的名称，它们的格式和之前例子中的类似，两个字符构成的语言代码（http://en.wikipedia.org/wiki/List_of_ISO_639-1_codes）加一个下划线和两个字符构成的国家代码（http://en.wikipedia.org/wiki/ISO_3166-1_alpha-2）：

```
>>> good_names = [name for name in names if \
len(name) == 5 and name[2] == '_']
```

看看前 5 个是什么？

```
>>> good_names[:5]
['sr_cs', 'de_at', 'nl_nl', 'es_ni', 'sp_yu']
```

如果想获取所有的德国语言，试试这个：

```
>>> de = [name for name in good_names if name.startswith('de')]
>>> de
['de_at', 'de_de', 'de_ch', 'de_lu', 'de_be']
```

10.4.4　其他模块

如果你觉得标准库模块不好用或者没有你想要的功能，可以试试下面这些第三方模块。

- `arrow`（http://crsmithdev.com/arrow/）
  这个模块有许多日期和时间函数，并提供了简单易用的 API。
- `dateutil`（http://labix.org/python-dateutil）
  这个模块可以解析绝大多数日期格式，并且可以很好地处理相对日期和时间。
- `iso8601`（https://pypi.python.org/pypi/iso8601）
  这个模块会完善标准库中对于 ISO8601 格式的处理。
- `fleming`（https://github.com/ambitioninc/fleming）
  这个模块提供了许多时区函数。

10.5　练习

(1) 把当前日期以字符串形式写入文本文件 today.txt。
(2) 从 today.txt 中读取字符串到 `today_string` 中。
(3) 从 `today_string` 中解析日期。
(4) 列出当前目录下的文件。
(5) 列出父目录下的文件。
(6) 使用 `multiprocessing` 创建三个进程，让它们等待随机的秒数（范围 1~5），打印出当前时间并退出。
(7) 用你的生日创建一个 `date` 对象。
(8) 你的生日是星期几？
(9) 你出生 10 000 天的日期是什么？

第11章

并发和网络

“时间是大自然防止所有事情同时发生的一种方法。空间是防止所有事情都发生在我身上的方法。”

——关于时间的名言（http://en.wikiquote.org/wiki/Talk:Time）

到目前为止，你写过的大多数程序都是在一个地方（一台机器）一次运行一行（顺序执行）。但是，我们可以在多个地方（分布式计算或者网络）同时做很多事（并发）。这样做有很多好处。

- 性能

  你的目标是让高速部件不间断运行，不用等待慢速部件。

- 鲁棒性

  数量可以提高安全性，可以同时执行多个相同任务，这样就不用担心出现硬件和软件错误。

- 简化

  最佳实践是把复杂任务分解成许多小任务，这样更容易创建、理解和修复。

- 通信

  发送数据并接收新数据本身就是一件非常有趣的事。

我们从并发开始，先用第 10 章中提到的非网络技术——进程和线程——构建一个程序。之后会介绍其他方法，比如回调、绿色线程和协程。最后会使用网络技术，仍然是从并发技术开始，逐步扩展到其他方面。

在编写本书时，本章介绍的一些 Python 包还没有兼容 Python 3。大多数情况下的示例代码只能在 Python 2 的解释器中运行，这个解释器我们称之为 `python2`。

11.1　并发

Python 官网基于标准库（https://docs.python.org/3/library/concurrency.html）介绍了常见的并发技术。页面中提到了很多包和技术，本章会介绍其中最有用的部分。

在计算机中，如果你的程序在等待，通常是因为以下两个原因。

- I/O 限制

  这个限制很常见。计算机的 CPU 速度非常快——比计算机内存快几百倍，比硬盘或者网络快几千倍。

- CPU 限制

  在处理数字运算任务时，比如科学计算或者图形计算，很容易遇到这个限制。

以下是和并发相关的两个术语。

- 同步

  一件事接着一件事发生，就像送葬队伍一样。

- 异步

  任务是互相独立的，就像派对参与者从不同的车上下来一样。

当你要用简单的系统和任务来处理现实中的问题时，迟早需要处理并发。假设你有一个网站，必须给用户很快地返回静态和动态网页。一秒是可以接受的，但是如果展示或者交互需要很长时间，用户就会失去耐心。谷歌和亚马逊的测试显示，页面加载速度降低一点就会导致流量大幅下降。

但是，如何处理需要长时间执行的任务呢，比如上传文件、改变图片大小或者查询数据库？显然无法用同步的 Web 客户端代码解决这个问题，因为同步就必然会产生等待。

在一台电脑中，如果你想尽快处理多个任务，就需要让它们互相独立。慢任务不应该阻塞其他任务。

10.3 节介绍了如何用多进程实现单机的并发工作。如果你需要改变图片的大小，Web 服务器代码可以调用一个独立、专用的图片处理进程，可以异步地并发执行。这样可以通过增加处理进程来横向扩展你的应用。

问题的关键是如何让这些进程并发地执行。任何共享控制或者状态管理都会导致瓶颈。另一个更大的问题是如何处理错误，因为并发计算比常规计算更难，更容易出现错误，因此成功的概率更低。

那么到底应该如何处理这些复杂的问题呢？我们来看一种优秀的多任务管理方法：*队列*。

11.1.1 队列

队列有点像列表：从一头添加事物，从另一头取出事物。这种队列被称为 FIFO（先进先出）。

假设你正在洗盘子，如果需要完成全部工作，需要洗每一个盘子、烘干并放好。你有很多种方法来完成这个任务。或许你会先洗第一个盘子，烘干并把它放好，之后用同样的方法来处理第二个盘子，以此类推。此外，你也可以执行批量操作，先洗完所有的盘子，再烘干所有的盘子，最后把它们都放好。这样做需要你有足够大的水池和烘干机来放置每一步积累的所有盘子。这些都是同步方法——一个工人，一次做一件事。

还有一种方法是再找一个或者两个帮手。如果你是洗盘子的人，可以把洗好的盘子递给烘干盘子的人，他再把烘干的盘子递给放置盘子的人。所有人都在自己的位置工作，这样会比你一个人要快很多。

然而，如果你洗盘子的速度比下一个人烘干的速度快怎么办？要么把湿盘子扔在地上，要么把它们堆在你和下一个人之间，或者一直闲着直到下一个人处理完之前的盘子。如果最后一个人比第二个人还慢，那第二个人要么把盘子扔在地上，要么把它们堆在两个人之间，要么就闲着。你有很多个工人，但总体来说，任务仍然是同步完成的，处理速度和最慢的工人速度是一样的。

俗话说：*人多好办事*（我总觉得这句话来自阿们宗派，因为它总会让我想到粮仓建筑）。增加工人可以更快地搭建粮仓或者洗盘子，前提是使用*队列*。

通常来说，队列用来传递消息，消息可以是任意类型的信息。在本例中，我们用队列来管理分布式任务，这种队列也称为*工作队列*或者*任务队列*。水池中的每个盘子都会发给一个闲置的洗盘子的人，他会洗盘子并把盘子传给第一个闲置的烘干盘子的人，他会烘干盘子并把盘子传给第一个闲置的放盘子的人。这个过程可以是同步的（工人等着处理盘子，处理完等着把盘子给下一个人），也可以是异步的（盘子堆在两个工人中间）。只要你有足够多的工人并且他们都能认真工作，完成速度会很快。

11.1.2 进程

可以用很多方法来实现队列。对单机来说，标准库中的 `multiprocessing` 模块（参见 10.3 节）有一个 `Queue` 函数。接下来模拟一个洗盘子的人和多个烘干进程（不用担心，之后会有人把这些盘子放好），我们使用一个中间队列 `dish_queue`。把下面的代码保存为 dishes.py：

```
import multiprocessing as mp

def washer(dishes, output):
    for dish in dishes:
        print('Washing', dish, 'dish')
        output.put(dish)

def dryer(input):
    while True:
        dish = input.get()
        print('Drying', dish, 'dish')
```

```
        input.task_done()

dish_queue = mp.JoinableQueue()
dryer_proc = mp.Process(target=dryer, args=(dish_queue,))
dryer_proc.daemon = True
dryer_proc.start()

dishes = ['salad', 'bread', 'entree', 'dessert']
washer(dishes, dish_queue)
dish_queue.join()
```

运行这个新程序：

```
$ python dishes.py
Washing salad dish
Washing bread dish
Washing entree dish
Washing dessert dish
Drying salad dish
Drying bread dish
Drying entree dish
Drying dessert dish
```

这个队列看起来很像一个简单的 Python 迭代器，会生成一系列盘子。这段代码实际上会启动几个独立的进程，洗盘子的人和烘干盘子的人会用它们来进行通信。我使用 JoinableQueue 和最后的 join() 方法让洗盘子的人知道，所有的盘子都已经烘干。multiprocessing 模块还有其他类型的队列，更多实例请参考文档（https://docs.python.org/3/library/multiprocessing.html）。

11.1.3　线程

线程运行在进程内部，可以访问进程的所有内容，有点像多重人格。multiprocessing 模块有一个兄弟模块 threading，后者用线程来代替进程（实际上，multiprocessing 是在 threading 之后设计出来的，基于进程来完成各种任务）。我们使用线程来重写一遍上面的进程示例：

```
import threading

def do_this(what):
    whoami(what)

def whoami(what):
    print("Thread %s says: %s" % (threading.current_thread(), what))

if __name__ == "__main__":
    whoami("I'm the main program")
    for n in range(4):
        p = threading.Thread(target=do_this,
          args=("I'm function %s" % n,))
        p.start()
```

运行后得到以下输出：

```
Thread <_MainThread(MainThread, started 140735207346960)> says: I'm the main
program
Thread <Thread(Thread-1, started 4326629376)> says: I'm function 0
Thread <Thread(Thread-2, started 4342157312)> says: I'm function 1
Thread <Thread(Thread-3, started 4347412480)> says: I'm function 2
Thread <Thread(Thread-4, started 4342157312)> says: I'm function 3
```

可以使用线程来重写上面的盘子示例：

```
import threading, queue
import time

def washer(dishes, dish_queue):
    for dish in dishes:
    print ("Washing", dish)
        time.sleep(5)
        dish_queue.put(dish)

def dryer(dish_queue):
    while True:
        dish = dish_queue.get()
        print ("Drying", dish)
        time.sleep(10)
        dish_queue.task_done()


dish_queue = queue.Queue()
for n in range(2):
    dryer_thread = threading.Thread(target=dryer, args=(dish_queue,))
        dryer_thread.start()

dishes = ['salad', 'bread', 'entree', 'desert']
washer(dishes, dish_queue)
dish_queue.join()
```

`multiprocessing` 和 `threading` 的区别之一就是 `threading` 没有 `terminate()` 函数。很难终止一个正在运行的线程，因为这可能会引起代码和时空连续性上的各种问题。

线程可能会很危险。就像 C 和 C++ 这类语言中的手动内存管理一样，线程可能会引起很难寻找和处理的 bug。要使用线程，程序中的所有代码——以及程序使用的所有外部库中的代码——必须是线程安全的。在之前的示例代码中，线程之间没有共享任何全局变量，因此可以在没有副作用的情况下独立运行。

假设你是一个幽灵屋中的超自然现象调查员，幽灵在大厅中漫游，但是它们互相之间并不能感知到对方。此外，幽灵可以在任意时间浏览、添加、删除或者移动房间中的任意物品。

你一边看着令人惊讶的仪表读数，一边穿过整个房间。突然，你发现几秒钟之前刚看过的烛台不见了。

房间中的物品就像程序中的变量，幽灵是进程（房间）中的线程。如果幽灵只会浏览房间中的物品，就没有任何问题。就像线程只会读取常量或者变量中的值，但是不会修改它们。

然而，有些看不见的东西会抓住你的手电筒，往你的脖子上吹冷风，在大理石楼梯上一步一步地走，或者点燃壁炉。真正精明的幽灵甚至会在你看不到的房间中捣乱。

尽管你有很高端的设备，要找出是谁在什么时候做了什么改动仍然非常困难。

如果使用进程来代替线程，那就像有很多个房子但是每个房子里只有一个（活）人。如果你把白兰地放在壁炉前，一个小时后它还会在那儿。或许会蒸发一些，但是位置不变。

没有全局变量时，线程是非常有用并且安全的。通常来说，如果需要等待 I/O 操作完成，那么使用线程可以节省很多时间。在这种情况下，线程不会因为数据打架，因为每个线程使用的是完全独立的变量。

但是线程有时候确实需要修改全局变量。实际上，使用多线程的一个常见目的就是把需要处理的数据进行划分，这就不可避免地需要修改数据。

常见的安全共享数据的方法是让线程在修改变量之前加软件锁，这样在进行修改时其他线程都会等待。这就像在有幽灵的房子中有一个抓幽灵敢死队帮你看门。需要注意的是，千万别忘了解锁。此外，锁可以嵌套，就像你还有另一个抓幽灵敢死队来看同一个房间或者同一个房子。锁的用法非常传统但是要想用对非常困难。

在 Python 中，线程不能加速受 CPU 限制的任务，原因是标准 Python 系统中使用了*全局解释器锁*（GIL）。GIL 的作用是避免 Python 解释器中的线程问题，但是实际上会让多线程程序运行速度比对应的单线程版本甚至是多进程版本更慢。

总而言之，对于 Python，建议如下：

- 使用线程来解决 I/O 限制问题；
- 使用进程、网络或者事件（下一节会介绍）来处理 CPU 限制问题。

11.1.4 绿色线程和gevent

如你所见，开发者通常会把程序中运行速度慢的部分划分为多个线程或者进程从而加快速度。Apache Web 服务器就是一个典型的例子。

另一种方法是基于事件编程。一个基于事件的程序会运行一个核心*事件循环*，分配所有任务，然后重复这个循环。nginx Web 服务器就是基于事件的设计，通常来说比 Apache 快。

`gevent` 就是一个基于事件的很棒的库：你只需要编写普通的代码，`gevent` 会神奇地把它们转换成*协程*。协程就像可以互相通信的生成器，它们会记录自己的位置。`gevent` 可以修改许多 Python 的标准对象，比如 `socket`，从而使用它自己的机制来代替阻塞。协程无法处理 C 写成的 Python 扩展代码，比如一些数据库驱动程序。

在编写本书时，`gevent` 还不能完全兼容 Python 3，因此下面的示例代码使用的是 Python 2 的工具 `pip2` 和 `python2`。

可以使用 Python 2 版的 pip 来安装 gevent：

```
$ pip2 install gevent
```

下面是 gevent 官网示例代码（http://www.gevent.org/）的一个变体。11.2.7 节会介绍 socket 模块的 gethostbyname() 函数。这个函数是同步的，所以当它在全世界的名称服务器中寻找地址时，你必须等待（可能要好几秒）。但是，你可以使用 gevent 版本的代码来同时查询多个网站的地址。把下面的代码保存为 gevent_test.py：

```
import gevent
from gevent import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(gevent.socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

这段代码中有一个只有一行的 for 循环。每个主机名都会被提交到一个 gethostbyname() 调用中，但是这些调用可以异步执行，因为使用的是 gevent 版本的 gethostbyname()。

使用下面的命令来用 Python 2 运行 gevent_test.py：

```
$ python2 gevent_test.py
66.6.44.4
74.125.142.121
78.136.12.50
```

gevent.spawn() 会为每个 gevent.socket.gethostbynome(url) 创建一个绿色线程（也叫微线程）。

绿色线程和普通线程的区别是前者不会阻塞。如果遇到会阻塞普通线程的情况，gevent 会把控制权切换到另一个绿色线程。

gevent.joinall() 方法会等待所有的任务完成。最后，我们会输出获得的所有 IP 地址。

除了使用 gevent 版本的 socket 之外，你也可以使用猴子补丁（monkey-patching）函数。这个函数会修改标准模块，比如 socket，直接让它们使用绿色线程而不是调用 gevent 版本。如果想在整个程序中应用 gevent，这种方法非常有用，即使那些你无法直接接触到的代码也会被改变。

在程序的开头，添加下面的代码：

```
from gevent import monkey
monkey.patch_socket()
```

这会把程序中所有的普通 socket 都修改成 gevent 版本，即使是标准库也不例外。再提醒一次，这个改动只对 Python 代码有效，对 C 写成的库无效。

另一个函数会给更多的标准库模块打上补丁：

```
from gevent import monkey
monkey.patch_all()
```

在程序开头加上这段代码可以让你的程序充分利用 gevent 带来的速度提升。

把下面的程序保存为 gevent_monkey.py：

```
import gevent
from gevent import monkey; monkey.patch_all()
import socket
hosts = ['www.crappytaxidermy.com', 'www.walterpottertaxidermy.com',
    'www.antique-taxidermy.com']
jobs = [gevent.spawn(socket.gethostbyname, host) for host in hosts]
gevent.joinall(jobs, timeout=5)
for job in jobs:
    print(job.value)
```

再使用 Python 2 运行程序：

```
$ python2 gevent_monkey.py
66.6.44.4
74.125.192.121
78.136.12.50
```

使用 gevent 还有一个潜在的危险。对于基于事件的系统来说，执行的每段代码都应该尽可能快。尽管不会阻塞，执行复杂任务的代码还是会很慢。

猴子补丁的理念对于很多人来说并不容易接受。但是，很多大型网站（比如 Pinterest）都在使用 gevent，对网站来说有明显的加速作用。就像一瓶外表精美的药丸一样，要用正确的方式使用 gevent。

> 另外两个流行的事件驱动框架是 tornado（http://www.tornadoweb.org/en/stable/）和 gunicorn（http://gunicorn.org/）。它们都使用了底层事件处理和高速 Web 服务器。如果你想使用传统的 Web 服务器（比如 Apache）来构建高速网站，这两个框架非常值得一看。

11.1.5　twisted

twisted（http://twistedmatrix.com/trac/）是一个异步事件驱动的网络框架。你可以把函数关联到事件（比如数据接收或者连接关闭）上，当事件发生时这些函数会被调用。这种设计被称为*回调*（callback），如果你以前用过 JavaScript，那一定不会陌生。如果是第一次见到回调，可能会觉得它有点过时。对于有些开发者来说，基于回调的代码在应用规模变大之后会很难维护。

和 gevent 一样，twisted 还没有兼容 Python 3。本节会使用 Python 2 的安装器和解释器。使用下面的命令来安装 twisted：

```
$ pip2 install twisted
```

twisted 很大，支持很多基于 TCP 和 UDP 的互联网协议。出于教学目的，我们会展示一个简单的敲门服务器和客户端，由 twisted 示例（http://twistedmatrix.com/documents/current/core/examples/）修改而来。首先来看服务器，把代码保存到 knock_server.py（注意 Python 2 中 print() 的语法）：

```
from twisted.internet import protocol, reactor

class Knock(protocol.Protocol):
    def dataReceived(self, data):
        print 'Client:', data
        if data.startswith("Knock knock"):
            response = "Who's there?"
        else:
            response = data + " who?"
        print 'Server:', response
        self.transport.write(response)

class KnockFactory(protocol.Factory):
    def buildProtocol(self, addr):
        return Knock()

reactor.listenTCP(8000, KnockFactory())
reactor.run()
```

现在看看服务器的忠实伙伴，knock_client.py：

```
from twisted.internet import reactor, protocol

class KnockClient(protocol.Protocol):
    def connectionMade(self):
        self.transport.write("Knock knock")

    def dataReceived(self, data):
        if data.startswith("Who's there?"):
            response = "Disappearing client"
            self.transport.write(response)
        else:
            self.transport.loseConnection()
            reactor.stop()

class KnockFactory(protocol.ClientFactory):
    protocol = KnockClient

def main():
    f = KnockFactory()
    reactor.connectTCP("localhost", 8000, f)
    reactor.run()

if __name__ == '__main__':
    main()
```

先启动服务器：

```
$ python2 knock_server.py
```

然后启动客户端：

```
$ python2 knock_client.py
```

服务器和客户端会交换信息，服务器打印以下对话内容：

```
Client: Knock knock
Server: Who's there?
Client: Disappearing client
Server: Disappearing client who?
```

之后，我们的魔术师客户端结束了，只留服务器还在不断等待。

如果想了解更多关于 twisted 的内容，可以试试官方文档中的其他示例。

11.1.6 asyncio

最近，吉多·范·罗苏姆（还记得他是谁吗？）参与处理 Python 的并发问题。许多包有自己的事件循环，每种事件循环都想成为标准。他该如何调停回调、绿色线程以及其他并发方法呢？经过许多讨论和交流，他发布了异步 IO 支持重新启动："asyncio" 模块（http://legacy.python.org/dev/peps/pep-3156/），代号 Tulip。asyncio 模块在 Python 3.4 中首次出现。目前，它提供了一种通用的事件循环，可以兼容 twisted、gevent 和其他异步方法。目标是提供一种标准、简洁、高性能的异步 API。期待它在未来的 Python 发布版中不断发展。

11.1.7 Redis

我们之前的洗盘子示例代码，无论使用的是进程还是线程，都运行在一台机器上。下面我们使用另一种方法来实现队列，让它可以既支持单机又支持网络。有时候用了进程和线程，单机仍然无法满足需求。本章的目的就是帮助你从一个盒子（单机）过渡到多个并发的盒子。

要运行本章的示例，需要安装 Redis 服务器和它的 Python 模块。安装方法参见 8.5.3 节。第 8 章中，Redis 的角色是数据库，而这里指的是它的并发特性。

可以使用 Redis 列表来快速创建一个队列。Redis 服务器部署在一台机器上；客户端可以部署在同一台机器上也可以部署在不同机器上，通过网络通信。无论是哪种情况，客户端都是通过使用 TCP 和服务器通信，因此它们是网络化的。一个或多个生产者客户端向列表的一端压入消息，一个或多个工人客户端通过阻塞弹出操作从列表中获得需要洗的盘子。如果列表为空，它们就会闲置。如果有一条消息，第一个空闲工人就会去处理。

和之前的基于进程和线程的示例一样，redis_washer.py 会生成一个盘子序列：

```
import redis
conn = redis.Redis()
print('Washer is starting')
dishes = ['salad', 'bread', 'entree', 'dessert']
for dish in dishes:
    msg = dish.encode('utf-8')
    conn.rpush('dishes', msg)
    print('Washed', num)
conn.rpush('dishes', 'quit')
print('Washer is done')
```

循环会生成四个包含盘子名称的消息，最后一条消息是"退出"（quit）。程序会把所有消息都添加到 Redis 服务器上的 dishes 列表中，就像添加到 Python 列表一样。

当第一个盘子就绪之后，redis_dryer.py 就开始工作了：

```
import redis
conn = redis.Redis()
print('Dryer is starting')
while True:
    msg = conn.blpop('dishes')
    if not msg:
        break
    val = msg[1].decode('utf-8')
    if val == 'quit':
        break
    print('Dried', val)
print('Dishes are dried')
```

这段代码会等待第一个令牌是 dishes 的消息，并打印出烘干的盘子。如果遇到“退出”消息就终止循环。

启动烘干工人，然后启动清洗工人。在命令结尾加上 &，让第一个程序在后台运行；它会一直运行下去，但是不会监听键盘。尽管第二行的输出稍有不同，但是这个技巧在 Linux、OS X 和 Windows 上都有效。在本例中（OS X），第二行输出的是和后台烘干进程相关的信息。接着，我们正常（在前台）启动清洗进程。你可以看到两个进程混在一起的输出：

```
$ python redis_dryer.py &
[2] 81691
Dryer is starting
$ python redis_washer.py
Washer is starting
Washed salad
Dried salad
Washed bread
Dried bread
Washed entree
Dried entree
Washed dessert
Washer is done
Dried dessert
Dishes are dried
[2]+  Done                    python redis_dryer.py
```

只要盘子 ID 从清洗进程到达 Redis，我们勤劳的烘干进程就会取出它们。每个盘子 ID 都是一个数字，除了最后的哨兵值，它是字符串 'quit'。当烘干进程读取到盘子 ID quit 就会退出，后台进程的信息会打印到终端（具体内容在不同系统中同样有差别）。你可以使用一个哨兵（或者说一个非法值）在数据流中表示一些特殊的意义——本例的意义是完毕。如果不这样做，需要添加一些编程逻辑，如下所示：

- 提前设定好盘子的最大值，这也是一种哨兵；
- 进行特殊的带外（不在数据流中）进程间通信；
- 一定时间没有新数据就退出。

再进行一些最终的修改：

- 创建多个 dryer 进程；

• 除了等待哨兵，给每个烘干进程添加一个超时时间。

新的 redis_dryer2.py：

```
def dryer():
    import redis
    import os
    import time
    conn = redis.Redis()
    pid = os.getpid()
    timeout = 20
    print('Dryer process %s is starting' % pid)
    while True:
        msg = conn.blpop('dishes', timeout)
        if not msg:
            break
        val = msg[1].decode('utf-8')
        if val == 'quit':
            break
        print('%s: dried %s' % (pid, val))
        time.sleep(0.1)
    print('Dryer process %s is done' % pid)

import multiprocessing
DRYERS=3
for num in range(DRYERS):
    p = multiprocessing.Process(target=dryer)
    p.start()
```

在后台启动烘干进程，接着在前台启动清洗进程：

```
$ python redis_dryer2.py &
Dryer process 44447 is starting
Dryer process 44448 is starting
Dryer process 44446 is starting
$ python redis_washer.py
Washer is starting
Washed salad
44447: dried salad
Washed bread
44448: dried bread
Washed entree
44446: dried entree
Washed dessert
Washer is done
44447: dried dessert
```

一个烘干进程读取 `quit` ID 并退出：

```
Dryer process 44448 is done
```

20 秒后，其他烘干进程的 `blpop` 调用返回 `None`，表示超时，所以它们打印出最后一句话并退出：

```
Dryer process 44447 is done
Dryer process 44446 is done
```

最后一个烘干子进程退出后，主烘干程序退出：

```
[1]+  Done                    python redis_dryer2.py
```

11.1.8　队列之上

加入的功能越多，流水线就越有可能出问题。如果需要给一个宴会洗盘子，工人数量是否足够呢？如果烘干工人喝多了怎么办？如果水槽堵了怎么办？好担心啊！

如何应对这一切呢？幸运的是，有三种技术可供你使用。

- *触发并忘记*

  只传递内容，并不关心结果，即使没人处理。这就是“把盘子扔地上”方法。

- *请求－响应*

  对于每一个盘子，流水线上的清洗工人需要收到烘干工人的确认，烘干工人需要收到放置工人的确认。

- *背压或者节流*

  适用于上游工人速度比下游工人快的情况。

在真实系统中，你必须保证工人的速度能够满足需求，否则就会听到盘子摔碎的声音。你可以把新任务添加到一个等待列表中，一些工人进程会从中弹出最后一个消息并把它添加到工作列表中。消息处理完成后会从工作列表中移除并被添加到完成列表。这样就可以知道哪些任务失败或者占用了太长的时间。你可以自己使用 Redis 来完成这些功能，或者使用其他人已经写好并通过测试的系统。以下有一些基于 Python 的队列包添加了这种额外的控制层（有些使用的是 Redis）。

- `celery`（http://www.celeryproject.org/）

  这个包非常值得一看。它可以同步或者异步执行分布式任务，使用了我们之前介绍的方法：`multiprocessing`、`gevent` 等。

- `thoonk`（https://github.com/andyet/thoonk.py）

  这个包基于 Redis 构建，可以创建任务队列并实现*发布－订阅*（下一节会介绍）。

- rq（http://python-rq.org/）

  这是一个处理任务队列的 Python 库，同样基于 Redis。

- Queues（http://queues.io/）

  这个网站介绍了队列化软件，其中有些是基于 Python 开发的。

11.2　网络

在讨论并发时，主要讨论的是时间：单机解决方案（进程、线程和绿色线程）。还简单介绍了网络化的解决方案（Redis、ZeroMQ）。现在，我们来单独介绍一下网络化，也就是跨空间的分布式计算。

11.2.1　模式

你可以使用一些基础的模式来搭建网络化应用。

最常见的模式是*请求－响应*，也被称为*客户端－服务器*。这个模式是同步的：用户会一直等待服务器的响应。在本书中，你已经看过了很多“请求－响应”的示例。你的 Web 浏览器也是一个客户端，向 Web 服务器发起一个 HTTP 请求并等待响应。

另一种常见的模式是*推送*或者*扇出*：你把数据发送到一个进程池中，空闲的工作进程会进行处理。一个典型的例子是有负载均衡的 Web 服务器。

和推送相反的是*拉取*或者*扇入*：你从一个或多个源接收数据。一个典型的例子是记录器，它会从多个进程接收文本信息并把它们写入一个日志文件。

还有一个和收音机或者电视广播很像的模式：*发布－订阅*。这个模式中，会有发送数据的发布者。在简单的发布－订阅系统中，所有的订阅者都会收到一份副本。更常见的情况是，订阅者只关心特定类型的数据（通常被称为*话题*），发布者只会发送这些数据。因此，和推送模式不同，可能会有超过一个订阅者收到数据。如果一个话题没有订阅者，相关的数据会被忽略。

11.2.2　发布–订阅模型

发布－订阅并不是队列，而是广播。一个或多个进程发布信息，每个订阅进程声明自己感兴趣的消息类型，然后每个消息都会被复制一份发给感兴趣的订阅进程。因此，一个消息可能只被处理一次，也可能多于一次，还可能完全不被处理。每个发布者只负责广播，并不知道谁（如果有人的话）在监听。

1. Redis

你可以使用 Redis 来快速搭建一个发布－订阅系统。发布者会发出包含话题和值的消息，订阅者会声明它们对什么话题感兴趣。

下面是发布者，redis_pub.py：

```
import redis
import random

conn = redis.Redis()
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
for msg in range(10):
    cat = random.choice(cats)
    hat = random.choice(hats)
    print('Publish: %s wears a %s' % (cat, hat))
    conn.publish(cat, hat)
```

每个话题是猫的一个品种，每个消息的值是帽子的一种类型。

下面是一个订阅者，redis_sub.py：

```
import redis
conn = redis.Redis()
```

```
topics = ['maine coon', 'persian']
sub = conn.pubsub()
sub.subscribe(topics)
for msg in sub.listen():
    if msg['type'] == 'message':
        cat = msg['channel']
        hat = msg['data']
        print('Subscribe: %s wears a %s' % (cat, hat))
```

订阅者只会展示猫的品种为 `'maine coon'` 或者 `'persian'` 的消息。`listen()` 方法会返回一个字典，如果它的类型是 `'message'`，那就是由发布者发出的消息。`'channel'` 键是话题（猫），`'data'` 键包含消息的值（帽子）。

如果你先启动发布者，这时没有订阅者，就像把一个哑剧演员扔到树林里一样（他会发出声音吗？），因此要先启动订阅者：

```
$ python redis_sub.py
```

接着启动发布者，它会发送 10 个消息，然后退出：

```
$ python redis_pub.py
Publish: maine coon wears a stovepipe
Publish: norwegian forest wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
Publish: maine coon wears a bowler
Publish: siamese wears a stovepipe
Publish: norwegian forest wears a tam-o-shanter
Publish: maine coon wears a bowler
Publish: persian wears a bowler
Publish: norwegian forest wears a bowler
Publish: maine coon wears a stovepipe
```

订阅者只关心两类猫：

```
$ python redis_sub.py
Subscribe: maine coon wears a stovepipe
Subscribe: maine coon wears a bowler
Subscribe: maine coon wears a bowler
Subscribe: persian wears a bowler
Subscribe: maine coon wears a stovepipe
```

我们并没有让订阅者退出，因此它会一直等待消息。如果重新启动一个发布者，那订阅者会继续抓取消息并输出。

可以使用任意数量的订阅者（和发布者）。如果一个消息没有订阅者，那它会从 Redis 服务器中消失。然而，如果有订阅者，消息会停留在服务器中，直到所有的订阅者都获取完毕。

2. ZeroMQ

还记得之前介绍过 ZeroMQ 的 `PUB` 和 `SUB` 套接字吗？终于轮到它们大显身手了。ZeroMQ 没有核心服务器，因此每个发布者都会发送给所有订阅者。我们来使用 ZeroMQ 重写一下上面的猫 – 帽子示例。发布者为 zmq_pub.py，内容如下所示：

```
import zmq
import random
import time
host = '*'
port = 6789
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
pub.bind('tcp://%s:%s' % (host, port))
cats = ['siamese', 'persian', 'maine coon', 'norwegian forest']
hats = ['stovepipe', 'bowler', 'tam-o-shanter', 'fedora']
time.sleep(1)
for msg in range(10):
    cat = random.choice(cats)
    cat_bytes = cat.encode('utf-8')
    hat = random.choice(hats)
    hat_bytes = hat.encode('utf-8')
    print('Publish: %s wears a %s' % (cat, hat))
    pub.send_multipart([cat_bytes, hat_bytes])
```

注意代码是如何用 UTF-8 来编码话题和值字符串的。

下面是订阅者 zmq_sub.py：

```
import zmq
host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
sub = ctx.socket(zmq.SUB)
sub.connect('tcp://%s:%s' % (host, port))
topics = ['maine coon', 'persian']
for topic in topics:
    sub.setsockopt(zmq.SUBSCRIBE, topic.encode('utf-8'))
while True:
    cat_bytes, hat_bytes = sub.recv_multipart()
    cat = cat_bytes.decode('utf-8')
    hat = hat_bytes.decode('utf-8')
    print('Subscribe: %s wears a %s' % (cat, hat))
```

在这段代码中，我们订阅了两个不同的比特值：用 UTF-8 编码的 `topics` 中的两个字符串。

> 这看起来有点过时，但是如果你想订阅所有话题，需要订阅空比特字符串 `b''`，否则什么消息都得不到。

注意，我们在发布者中调用了 `send_multipart()`，在订阅者中调用了 `recv_multipart()`。这样就可以收到消息的多个部分并使用第一部分来判断话题是否匹配。也可以选择使用一个字符串或者比特字符串来发送话题和消息值，但是把猫和帽子分开发送会更加清晰。

启动订阅者：

```
$ python zmq_sub.py
```

启动发布者，它会立刻发送 10 条消息并退出：

```
$ python zmq_pub.py
Publish: norwegian forest wears a stovepipe
Publish: siamese wears a bowler
Publish: persian wears a stovepipe
Publish: norwegian forest wears a fedora
Publish: maine coon wears a tam-o-shanter
Publish: maine coon wears a stovepipe
Publish: persian wears a stovepipe
Publish: norwegian forest wears a fedora
Publish: norwegian forest wears a bowler
Publish: maine coon wears a bowler
```

订阅者打印出它想要的内容：

```
Subscribe: persian wears a stovepipe
Subscribe: maine coon wears a tam-o-shanter
Subscribe: maine coon wears a stovepipe
Subscribe: persian wears a stovepipe
Subscribe: maine coon wears a bowler
```

3. 其他发布-订阅工具

你可能会对 Python 的其他发布 – 订阅工具感兴趣。

- RabbitMQ

  这是一个非常著名的消息发送器。`pika` 是它的 Python API。详情参见 `pika` 文档（http://pika.readthedocs.org/）和发布 – 订阅教程（http://www.rabbitmq.com/tutorials/tutorial-three-python.html）。

- `pypi.python.org`

  在右上角的搜索框内输入 `pubsub` 来寻找类似 `pypubsub`（http://pubsub.sourceforge.net/）这样的 Python 包。

- `pubsubhubbub`

  这个读起来非常顺口的协议（https://code.google.com/p/pubsubhubbub/）允许订阅者注册对应发布者的回调函数。

11.2.3　TCP/IP

我们一直处在网络的世界中，理所当然地认为底层的一切都可以正常工作。现在，我们来真正地深入底层，看看那些维持系统运转的东西到底是什么样。

因特网是基于规则的，这些规则定义了如何创建连接、交换数据、终止连接、处理超时等。这些规则被称为协议，它们分布在不同的层中。分层的目的是兼容多种实现方法。你可以在某一层中做任何想做的事，只要遵循上一层和下一层的约定就行。

最底层处理的是电信号，其余层都基于下面的层构建而成。在大约中间的位置是 IP（因特网协议）层，这层规定了网络位置和地址的映射方法以及数据包（块）的传输方式。IP 层的上一层有两个协议描述了如何在两个位置之间移动比特。

- UDP（*用户数据报协议*）

  这个协议被用来进行少量数据交换。一个*数据报*是一次发送的很少信息，就像明信片上的一个音符一样。

- TCP（*传输控制协议*）

  这个协议被用来进行长时间的连接。它会发送比特流并确保它们都能按序到达并且不会重复。

UDP 信息并不需要确认，因此你永远无法确认它是否到达目的地。如果你想讲一个 UDP 笑话：

```
Here's a UDP joke. Get it?(这是一个UDP笑话,你笑了吗？)
```

TCP 会在发送者和接收者之间通过秘密握手建立有保障的连接。下面是一个 TCP 笑话：

```
Do you want to hear a TCP joke?(你想听一个TCP笑话吗？)
Yes, I want to hear a TCP joke.(是的,我想听一个TCP笑话。)
Okay, I'll tell you a TCP joke.(好的,我会告诉你一个TCP笑话。)
Okay, I'll hear a TCP joke.(好的,我会听到一个TCP笑话。)
Okay, I'll send you a TCP joke now.(好的,我现在要发给你一个TCP笑话。)
Okay, I'll receive the TCP joke now.(好的,我现在会收到一个TCP笑话。)
... (and so on)(下面省略)
```

你的本地机器 IP 地址一直是 `127.0.0.1`，名称一直是 `localhost`。你可能听过它的另一个名字*环回接口*。如果连接到因特网，那你的机器还会有一个公共 IP。如果使用的是家用计算机，那它一般会接到调制解调器或者路由器上。你甚至可以在同一台机器的两个进程之间使用因特网协议。

在因特网上，我们接触到的大多数事物——Web、数据库服务器，等等——都是基于 IP 协议上的 TCP 协议运行的。简单起见，写为 TCP/IP。下面先来看一些基本的因特网服务，然后会了解一些常用的网络化模式。

11.2.4 套接字

我们一直把这个话题留到现在才讲，是因为即使你不知道所有的底层细节也可以使用高层的因特网。但是，如果你想知道底层的工作原理，那就读读这节吧。

最底层的网络编程使用的是*套接字*，源于 C 语言和 Unix 操作系统。套接字层的编程是非常繁琐的。使用类似 ZeroMQ 的库会简单很多，但是了解一下底层的工作原理还是非常有用的。举例来说，网络发生错误时出现的错误信息通常是和套接字相关的。

我们来编写一个非常简单的客户端 - 服务器通信示例。客户端发送一个包含字符串的 UDP 数据报给服务器，服务器会返回一个包含字符串的数据包。服务器需要监听特定的地址和端口——就像邮局和邮筒一样。客户端需要知道这两个值才能发送、接收和响应消息。

在下面的客户端和服务器代码中，`address` 是一个（*地址，端口*）元组。`address` 是一个字符串，可以是名称或者 *IP 地址*。当你的程序和同一台机器上的另一个程序通信时，可以使用名称 `'localhost'` 或者等价的地址 `'127.0.0.1'`。

首先从一个进程给另一个进程发送一些数据，让后者返回一些数据。第一个程序是客户

端，第二个程序是服务器。在这两个程序中，我们都会打印出时间并打开一个套接字。服务器会监听它套接字上的连接，客户端会向它的套接字写入数据，套接字会发送一个消息给服务器。

下面是第一个程序，udp_server.py：

```
from datetime import datetime
import socket

server_address = ('localhost', 6789)
max_size = 4096

print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(server_address)

data, client = server.recvfrom(max_size)

print('At', datetime.now(), client, 'said', data)
server.sendto(b'Are you talking to me?', client)
server.close()
```

服务器必须用 `socket` 包中的两个方法来建立网络连接。第一个方法是 `socket.socket`，它会创建一个套接字。第二个方法 `bind` 会绑定（监听到达这个 IP 地址和端口的所有数据）到这个套接字上。`AF_INET` 表示要创建一个因特网（IP）套接字。（还有其他类型的 Unix 域套接字，不过那些只能在本地运行。）`SOCK_DGRAM` 表示我们要发送和接收数据报，换句话说，我们要使用 UDP。

之后，服务器会等待数据报到达（`recvfrom`）。收到数据报后，服务器会被唤醒并获取数据和客户端的信息。`client` 变量包含客户端的地址和端口，用于给客户端发送数据。接着，服务器发送一个响应并关闭连接。

下面，我们来看看 udp_client.py：

```
import socket
from datetime import datetime

server_address = ('localhost', 6789)
max_size = 4096

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.sendto(b'Hey!', server_address)
data, server = client.recvfrom(max_size)
print('At', datetime.now(), server, 'said', data)
client.close()
```

客户端的许多方法和服务器一样（除了 `bind()`）。客户端先发送数据，然后接收数据，而服务器恰好相反。

先在一个窗口中启动服务器。它会打印出欢迎信息，然后一直沉默，直到客户端发送数据：

```
$ python udp_server.py
Starting the server at 2014-02-05 21:17:41.945649
Waiting for a client to call.
```

接着在另一个窗口中启动客户端。它会打印出欢迎信息并向服务器发送数据，打印出响应并退出：

```
$ python udp_client.py
Starting the client at 2014-02-05 21:24:56.509682
At 2014-02-05 21:24:56.518670 ('127.0.0.1', 6789) said b'Are you talking to me?'
```

最后，服务器会打印类似下面的内容并退出：

```
At 2014-02-05 21:24:56.518473 ('127.0.0.1', 56267) said b'Hey!'
```

客户端需要知道服务器的地址和端口号，但是并不需要指定自己的端口号。它的端口号由系统自动分配——在本例中是 56267。

UDP 使用一个块来发送数据，并且不能保证一定可以送达。如果你使用 UDP 发送多个消息，那它们可能以任何顺序到达，也有可能全部都无法到达。UDP 很快、很轻，不需要建立连接，但是并不可靠。

由于 UDP 不可靠，我们准备使用 TCP（传输控制协议）。TCP 用来进行长时间连接，比如 Web。TCP 按照发送的顺序传输数据。如果出现任何问题，它会尝试重新传输。我们尝试一下使用 TCP 在客户端和服务器之间传输一些包。

tcp_client.py 和之前的 UDP 客户端有点像，只向服务器发送一个字符串，但是在调用套接字时有一些区别，如下所示：

```
import socket
from datetime import datetime

address = ('localhost', 6789)
max_size = 1000

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect(address)
client.sendall(b'Hey!')
data = client.recv(max_size)
print('At', datetime.now(), 'someone replied', data)
client.close()
```

我们把 SOCK_DGRAM 换成了 SOCK_STREAM，指定使用流协议 TCP。还使用 connect() 来建立流。使用 UDP 时不需要这么做，因为每个数据报都是直接暴露在互联网中。

tcp_server.py 和 UDP 版本也有一些区别：

```
from datetime import datetime
import socket

address = ('localhost', 6789)
```

```
max_size = 1000

print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind(address)
server.listen(5)

client, addr = server.accept()
data = client.recv(max_size)

print('At', datetime.now(), client, 'said', data)
client.sendall(b'Are you talking to me?')
client.close()
server.close()
```

`server.listen(5)` 的意思是最多可以和 5 个客户端连接，超过 5 个就会拒绝。`server.accept()` 接收第一个到达的消息，`client.recv(1000)` 指定最大的可接收消息长度为 1000 字节。

和之前一样，先启动服务器再启动客户端，然后看看会发生什么。首先是服务器：

```
$ python tcp_server.py
Starting the server at 2014-02-06 22:45:13.306971
Waiting for a client to call.
At 2014-02-06 22:45:16.048865 <socket.socket object, fd=6, family=2, type=1,
    proto=0> said b'Hey!'
```

接着是客户端。它会给服务器发送消息、接收响应并退出：

```
$ python tcp_client.py
Starting the client at 2014-02-06 22:45:16.038642
At 2014-02-06 22:45:16.049078 someone replied b'Are you talking to me?'
```

服务器会收集消息、打印出来、发送响应然后退出：

```
At 2014-02-06 22:45:16.048865 <socket.socket object, fd=6, family=2, type=1,
    proto=0> said b'Hey!'
```

可以看到，TCP 服务器使用 `client.sendall()` 发送响应，之前的 UDP 服务器使用的是 `client.sendto()`。TCP 会维持多个客户端 – 服务器套接字并保存客户端的 IP 地址。

这看起来并不坏，但是如果你试着编写更复杂的代码，那就会体会到套接字有多难写。下面是一些需要处理的问题。

- UDP 可以发送消息，但是消息的大小有限制，而且不能保证消息到达目的地。
- TCP 发送字节流，不是消息。你不知道每次调用时系统会发送或者接收多少字节。
- 如果要用 TCP 传输完整的消息，需要一些额外的信息来把片段拼凑成整个消息：固定的消息大小（字节）、整个消息的大小或者一些特殊的哨兵字符。
- 由于消息是字节，不是 Unicode 文本字符串，你需要使用 Python 的 `bytes` 类型。更多内容参见第 7 章。

看完这些之后，如果你还对套接字编程感兴趣，可以看看 Python 套接字编程教程（https://docs.python.org/3/howto/sockets.html）。

11.2.5 ZeroMQ

我们已经看过如何用 ZeroMQ 套接字创建发布 - 订阅模型。ZeroMQ 是一个库，有时候也被称为打了激素的套接字（sockets on steroids），ZeroMQ 套接字实现了很多你需要但是普通套接字没有的功能：

- 传输完整的消息
- 重连
- 当发送方和接收方的时间不同步时缓存数据

这个在线教程（http://zguide.zeromq.org/）写得很好，是我见过的最好的讲解网络化模型的教程。印刷版（*ZeroMQ: Messaging for Many Applications*，Pieter Hintjens 著，O'Reilly 出版社）中的代码风格很好，封面上还有一条大鱼。印刷版中的示例都是用 C 语言写成的，但是在线版可以选择很多种语言，比如 Python 版示例（https://github.com/imatix/zguide/tree/master/examples/Python）。本章会介绍一些 Python 写成的简单的 ZeroMQ 示例。

ZeroMQ 就像乐高积木，我们都知道用很少的乐高积木就能搭建出很多东西。在本例中，你可以用很少几个套接字类型和模式来构建网络。下面这些"乐高积木块"是 ZeroMQ 的套接字类型，看起来很像之前说过的网络模型：

- REQ（同步请求）
- REP（同步响应）
- DEALER（异步请求）
- ROUTER（异步响应）
- PUB（发布）
- SUB（订阅）
- PUSH（扇出）
- PULL（扇入）

在动手尝试之前，需要先安装 Python 的 ZeroMQ 库：

```
$ pip install pyzmq
```

最简单的模式是一个请求 – 响应对。这是同步的：一个套接字发送请求，另一个发送响应。首先是发送响应的代码（服务器），zmq_server.py：

```
import zmq

host = '127.0.0.1'
port = 6789
context = zmq.Context()
server = context.socket(zmq.REP)
server.bind("tcp://%s:%s" % (host, port))
while True:
    #  等待客户端的下一个请求
    request_bytes = server.recv()
    request_str = request_bytes.decode('utf-8')
    print("That voice in my head says: %s" % request_str)
    reply_str = "Stop saying: %s" % request_str
```

```
        reply_bytes = bytes(reply_str, 'utf-8')
        server.send(reply_bytes)
```

创建一个 Context 对象：这是一个能够保存状态的 ZeroMQ 对象。接着创建一个 REP（用于响应）类型的 ZeroMQ 套接字。调用 bind()，让它监听特定的 IP 地址和端口。注意，地址和端口用字符串 'tcp://localhost:6789' 来指定，并不是普通套接字中的元组。

这个示例代码会从一个发送者接收请求并发送响应。消息可以非常长——ZeroMQ 会处理这些细节。

下面是对应的请求代码（客户端），zmq_client.py。它的类型是 REQ（用于请求），而且调用的是 connect()，不是 bind()：

```
import zmq

host = '127.0.0.1'
port = 6789
context = zmq.Context()
client = context.socket(zmq.REQ)
client.connect("tcp://%s:%s" % (host, port))
for num in range(1, 6):
    request_str = "message #%s" % num
    request_bytes = request_str.encode('utf-8')
    client.send(request_bytes)
    reply_bytes = client.recv()
    reply_str = reply_bytes.decode('utf-8')
    print("Sent %s, received %s" % (request_str, reply_str))
```

现在是时候启动它们了。和普通套接字不同的一点是，你可以用任何顺序启动服务器和客户端。在后台的一个窗口中启动服务器：

```
$ python zmq_server.py &
```

然后在同一个窗口中启动客户端：

```
$ python zmq_client.py
```

你会看到客户端和服务器交替输出如下所示的内容：

```
That voice in my head says 'message #1'
Sent 'message #1', received 'Stop saying message #1'
That voice in my head says 'message #2'
Sent 'message #2', received 'Stop saying message #2'
That voice in my head says 'message #3'
Sent 'message #3', received 'Stop saying message #3'
That voice in my head says 'message #4'
Sent 'message #4', received 'Stop saying message #4'
That voice in my head says 'message #5'
Sent 'message #5', received 'Stop saying message #5'
```

客户端发送完第五条消息之后就退出了，但是我们并没有让服务器退出，所以它一直在等待消息。如果再次运行客户端，它会打印出相同的五行，服务器也会打印出这五行。如果不终止 zmq_server.py 进程并且再次运行它，那 Python 会抱怨说地址已经被使用：

```
$ python zmq_server.py &

[2] 356
Traceback (most recent call last):
  File "zmq_server.py", line 7, in <module>
    server.bind("tcp://%s:%s" % (host, port))
  File "socket.pyx", line 444, in zmq.backend.cython.socket.Socket.bind
      (zmq/backend/cython/socket.c:4076)
  File "checkrc.pxd", line 21, in zmq.backend.cython.checkrc._check_rc
      (zmq/backend/cython/socket.c:6032)
zmq.error.ZMQError: Address already in use
```

消息需要用字节字符串形式发送，所以需要把示例中的字符串用 UTF-8 格式编码。你可以发送任意类型的消息，只要把它转换成 bytes 就行。我们的消息是简单的文本字符串，所以 encode() 和 decode() 可以实现文本字符串和字节字符串的转换。如果你的消息包含其他数据类型，可以使用类似 MessagePack（http://msgpack.org/）的库来处理。

由于任何数量的 REQ 客户端都可以 connect() 到一个 REP 服务器，即使是基础的请求 - 响应模式也可以实现一些有趣的通信模式。服务器是同步的，一次只能处理一个请求，但是并不会丢弃这段时间到达的其他请求。ZeroMQ 会在触发某些限制之前一直缓存这些消息，直到它们被处理；这就是 ZeroMQ 中 Q 的意思。Q 表示队列（Queue），M 表示消息（Message），Zero 表示不需要任何消息分发者。

虽然 ZeroMQ 不需要任何核心分发者（中间人），但是如果需要，你可以搭建一个。举例来说，可以使用 DEALER 和 ROUTER 套接字异步连接到多个源和 / 或目标。

多个 REQ 套接字可以连接到一个 ROUTER 上，后者会把请求传递给 DEALER，DEALER 又会传递给和它连接的所有 REP 套接字（图 11-1）。就像很多浏览器连接到一个代理服务器，后者连接到一个 Web 服务器群。你可以根据需要添加任意数量的客户端和服务器。

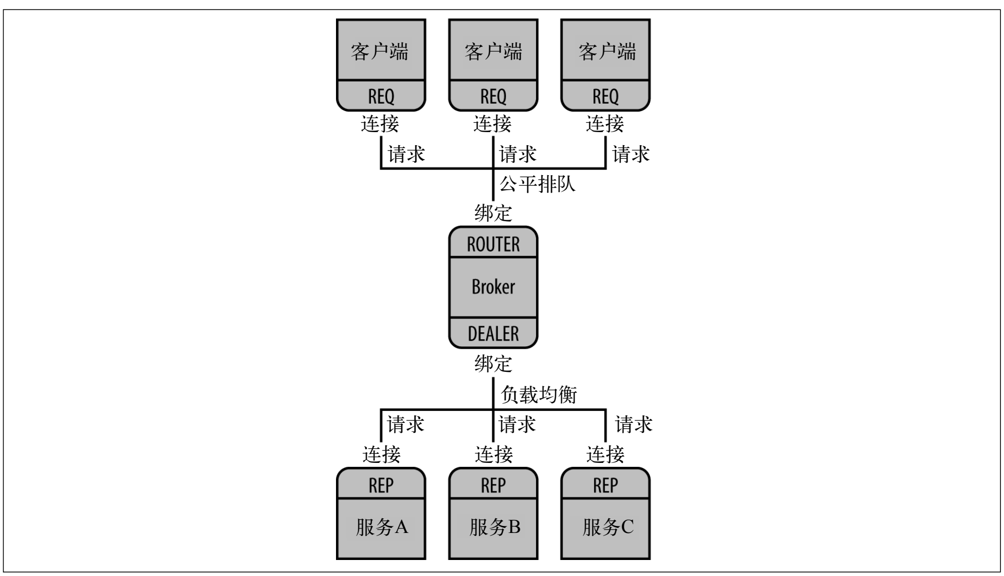


图 11-1：使用一个分发者连接多个客户端和服务器

REQ 套接字只能和 ROUTER 套接字连接；DEALER 可以和后面的多个 REP 套接字连接。ZeroMQ 会处理具体的细节部分，确保请求的负载均衡并把响应发送给正确的目标。

另一种网络化模式被称为通风口，使用 PUSH 套接字来发送异步任务，使用 PULL 套接字来收集结果。

最后一个需要介绍的 ZeroMQ 特性是它可以实现扩展和收缩，只要改变创建的套接字连接类型即可：

- `tcp` 适用于单机或者分布式的进程间通信
- `ipc` 适用于单机的进程间通信
- `inproc` 适用于单个进程内线程间通信

最后的 `inproc` 是一种线程间无锁的数据传输方式，可以替代 11.1.3 节中的 `threading` 示例。

使用 ZeroMQ 之后，你应该再也不会想写原始的套接字代码了。

> ZeroMQ 并不是 Python 支持的唯一一个消息传递库。消息传递是网络化的一个重要内容，Python 当然也不能落后。9.2.6 节中“Apache”部分提到的 Web 服务器是 Apache 项目的一部分，这个项目也在维护 ActiveMQ（https://activemq.apache.org/）项目，其中包含了几个使用简单文本 STOMP（http://stomp.github.io/implementations.html）协议的 Python 接口。RabbitMQ（http://www.rabbitmq.com/）也很出名，并且有优秀的 Python 教程（http://www.rabbitmq.com/tutorials/tutorial-one-python.html）。

11.2.6 scapy

有时候你需要深入网络流中处理字节。你可能想要调试 Web API 或者追踪一些安全问题。`scapy` 库是一个优秀的 Python 数据包分析工具，比编写和调试 C 程序简单很多。实际上，它是一门简单的用来构建和分析数据包的语言。

我本来计划在这里展示一些示例代码，但是由于以下两点原因改变了想法。

- `scapy` 还不兼容 Python 3。这个问题之前我们也遇到过，当时是使用 `pip2` 和 `python2` 来解决，但是……
- 在我看来，`scapy` 的安装教程（http://www.secdev.org/projects/scapy/portability.html）对于一本入门级的书来说太复杂了。

如果你愿意，可以看看文档（http://www.secdev.org/projects/scapy/doc/）中的示例代码。这些代码可能会让你有勇气在机器上安装 `scapy`。

最后，不要把 `scapy` 和 `crapy` 搞混，后者在 9.3.4 节有介绍。

11.2.7 网络服务

Python 有许多网络工具。下面的内容会介绍如何用自动化的方式实现那些最流行的网络服

务。官方的完整文档（https://docs.python.org/3/library/internet.html）可以在线查看。

1. 域名系统

计算机有类似 `85.2.101.94` 的数字 IP 地址，但是相比数字，我们更容易记住名称。域名系统（DNS）是一个非常重要的网络服务，通过一个分布式的数据库实现 IP 地址和名称的转换。当你使用 Web 浏览器并且看到类似"查找主机"的消息时，那可能就是网络连接中断了，第一种可能就是 DNS 错误。

在底层 `socket` 模块中有一些 DNS 函数。`gethostbyname()` 会返回一个域名的 IP 地址、扩展版本 `gethostbyname_ex()` 会返回名称、一个可选名称列表和一个地址列表：

```
>>> import socket
>>> socket.gethostbyname('www.crappytaxidermy.com')
'66.6.44.4'
>>> socket.gethostbyname_ex('www.crappytaxidermy.com')
('crappytaxidermy.com', ['www.crappytaxidermy.com'], ['66.6.44.4'])
```

`getaddrinfo()` 方法会查找 IP 地址，不过它返回的信息很全，可以用于创建套接字连接：

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80)
[(2, 2, 17, '', ('66.6.44.4', 80)), (2, 1, 6, '', ('66.6.44.4', 80))]
```

上面的调用会返回两个元组，第一个用于 UDP，第二个用于 TCP（`2, 1, 6` 中的 `6` 表示的就是 TCP）。

你可以只获取 TCP 或者 UDP 信息：

```
>>> socket.getaddrinfo('www.crappytaxidermy.com', 80, socket.AF_INET,
socket.SOCK_STREAM)
[(2, 1, 6, '', ('66.6.44.4', 80))]
```

有些 TCP 和 UDP 端口号（http://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers）是 IANA 为特定服务保留的，每个端口号关联一个服务名。举例来说，HTTP 的名称是 `http`，关联到 TCP 端口 80。

下面的函数可以实现服务名和端口号的转换：

```
>>> import socket
>>> socket.getservbyname('http')
80
>>> socket.getservbyport(80)
'http'
```

2. Python的Email模块

标准库中有以下这些 Email 模块：

- `smtplib` 使用简单邮件传输协议（SMTP）发送邮件；
- `email` 用来创建和解析邮件；
- `poplib` 可以使用邮递协议（POP3）来读取邮件；
- `imaplib` 可以使用因特网消息访问协议（IMAP）来读取邮件。

官方文档包含这些库对应的示例代码（https://docs.python.org/3/library/email-examples.html）。

如果你想编写自己的 Python SMTP 服务器，可以试试 `smtpd`（https://docs.python.org/3/library/smtpd.html）。

Lamson（http://Lamsonproject.org/）是一个纯 Python 的 SMTP 服务器，可以在数据库中存储邮件，甚至可以过滤垃圾邮件。

3. 其他协议

标准的 `ftplib` 模块（https://docs.python.org/3/library/ftplib.html）可以使用文件传输协议（FTP）来发送字节。虽然这是一个很古老的协议，但它的表现仍然非常优秀。

本书已经介绍了很多标准库中的模块，不过还是推荐你阅读一下标准库文档中的网络协议（https://docs.python.org/3/library/internet.html）部分。

11.2.8 Web服务和API

信息提供商都有网站，但是这些网站的目标是普通用户，并不是自动化。如果数据只展示在网页上，那想要获取并结构化这些数据的人就必须编写爬虫（参见 9.3.4 节），在页面格式改变时还必须更新爬虫。这是一件很麻烦的事。但是如果一个网站提供数据 API，那对于客户端程序来说，数据的获取就会变得非常直观。相比网页布局，API 很少改变，因此客户端也不需要经常重写。一个快速、整洁的数据通道可以大大简化*混搭程序*的编写难度——这些程序虽然不具备前瞻性，但是可能非常有用并且能带来利润。

通常来说，最简单的 API 是一个 Web 接口，可以提供类似 JSON 或者 XML 的结构化数据，而不是纯文本或者 HTML。API 既可以做得非常简单也可以是一套成熟的 RESTful API（9.3.2 节有具体定义），后者可以更好地处理那些不安分的字节[1]。

在本书的最开始，你就看过一个 Web API：它从 YouTube 上获取最流行的视频。在看过 Web 请求、JSON、字典、元组和切片之后，这个例子应该已经很容易理解了：

```
import requests
url = "https://gdata.youtube.com/feeds/api/standardfeeds/top_rated?alt=json"
response = requests.get(url)
data = response.json()
for video in data['feed']['entry'][0:6]:
    print(video['title']['$t'])
```

在挖掘知名社交媒体网站，比如 Twitter、Facebook 和 LinkedIn 时，API 非常有用。这些网站都提供可以免费使用的 API，但是它们要求你注册来获得一个 key（一个很长的文本字符串，有时也被称为 token），使用这个 key 来访问 API。网站可以通过 key 来判断是谁在获取数据，也可以用它来限制请求频率。在 YouTube 这个例子中，进行搜索不需要 API key，但是如果要更新 YouTube 上的数据，那就必须使用 key。

下面是一些有趣的服务 API：

- 纽约时报（http://developer.nytimes.com/）
- YouTube（http://gdata.youtube.com/demo/index.html）

注 1：英语“不安分”是 restless，正好和 RESTful 对应。——译者注

- Twitter（https://dev.twitter.com/docs/twitter-libraries）
- Facebook（https://developers.facebook.com/tools/）
- Weather Underground（http://www.wunderground.com/weather/api/）
- 漫威漫画（http://developer.marvel.com/）

你可以在附录 B 中看到地图 API 示例，附录 C 中还有其他示例。

11.2.9　远程处理

本书中的很多示例都是介绍如何在同一台机器上调用 Python 代码，通常还是在同一个进程中。但是 Python 的能力远不止这些，你可以调用其他机器上的代码，就像它们在本地一样。在高级设置中，如果你用完了单机的空间，可以扩展到其他机器。通过网络连接的一组计算机可以让你操作更多进程和 / 或线程。

1. 远程过程调用

远程过程调用（RPC）看起来和普通的函数一样，但其实运行在通过网络连接的远程机器上。RESTful API 需要通过 URL 编码参数或者请求体来调用，但是 RPC 函数是在你自己的机器上调用。下面是 RPC 客户端的工作原理：

(1) 把你的函数参数转换成比特（有时候被称为编组、序列化或者编码）；
(2) 把编码后的字节发送给远程机器。

下面是远程机器的工作原理：

(1) 接收编码后的请求字节；
(2) 接收完毕后，RPC 客户端会把字节解码成原始的数据结构（或者等价的东西，如果两台机器的硬件和软件有差别）；
(3) 客户端找到本地目标函数并用解码后的数据调用它；
(4) 客户端编码函数执行结果；
(5) 客户端把编码后的字节发送给调用者。

最后，发起请求的机器把字节解码成返回值。

RPC 是一种非常流行的技术，有很多种实现方式。在服务端，你可以启动一个服务器程序，把它和一些字节传输和编码 / 解码方法连接起来，定义一些访问函数并宣布你的 RPC 开始正常运转。客户端可以连接到服务器并通过 RPC 调用服务器的函数。

标准库中包含一种 RPC 实现，`xmlrpc`，使用 XML 作为传输格式。你在服务器上定义并注册函数，客户端使用类似导入的方式来调用它们。首先来看文件 xmlrpc_server.py：

```
from xmlrpc.server import SimpleXMLRPCServer

def double(num):
    return num * 2

server = SimpleXMLRPCServer(("localhost", 6789))
server.register_function(double, "double")
server.serve_forever()
```

我们在服务器上提供了 double() 函数，它接收一个数字参数并返回这个数字乘以 2 的结果。服务器在一个地址和端口上启动。我们需要注册函数，这样它才能让客户端通过 RPC 调用。最后，启动服务器并等待。

接着，和你想的一样，xmlrpc_client.py：

```
import xmlrpc.client

proxy = xmlrpc.client.ServerProxy("http://localhost:6789/")
num = 7
result = proxy.double(num)
print("Double %s is %s" % (num, result))
```

客户端通过 ServerProxy() 和服务器连接。接着它会调用 proxy.double()。这个函数是哪儿来的？实际上，它是由服务器动态生成的。RPC 机制会截获这个函数名并在远程服务器上调用它。

下面来看一下效果，先启动服务器：

```
$ python xmlrpc_server.py
```

接着启动客户端：

```
$ python xmlrpc_client.py
Double 7 is 14
```

服务器会打印出如下内容：

```
127.0.0.1 - - [13/Feb/2014 20:16:23] "POST / HTTP/1.1" 200 -
```

常用的传输方式是 HTTP 和 ZeroMQ。除了 XML 外，JSON、Protocol Buffers 和 MessagePack 也是常用的编码方式。有许多基于 JSON 的 Python RPC 包，但是它们要么不支持 Python 3，要么太难用。这里我们使用 MessagePack 自己的 Python RPC 实现（https://github.com/msgpack-rpc/msgpack-rpc-python）。下面是安装方法：

```
$ pip install msgpack-rpc-python
```

这条命令还会安装 tornado，这是一个基于事件的 Python Web 服务器，会被这个库用于传输数据。按照惯例，先是服务器的代码（msgpack_server.py）：

```
from msgpackrpc import Server, Address

class Services():
    def double(self, num):
        return num * 2

server = Server(Services())
server.listen(Address("localhost", 6789))
server.start()
```

Services 类把它的方法暴露为 RPC 服务。下面是客户端 msgpack_client.py：

```
from msgpackrpc import Client, Address
```

```
client = Client(Address("localhost", 6789))
num = 8
result =  client.call('double', num)
print("Double %s is %s" % (num, result))
```

依照惯例，先启动服务器再启动客户端：

```
$ python msgpack_server.py
```

```
$ python msgpack_client.py
Double 8 is 16
```

2. fabric

fabric 包可以运行远程或者本地命令、上传或者下载文件、用 sudo 权限运行命令。这个包使用安全 Shell（SSH：加密文本协议，基本上已经代替了 telnet）来运行远程程序。你需要把（Python）函数写入一个 fabric 文件并声明它们应该在远程还是本地执行。之后，使用时需要用 fabric 程序（名字是 fab，但是并不是向披头士或者洗涤灵致敬）来运行，需要指定目标远程机器和目标函数。它比 RPC 简单很多。

> 在编写本书时，fabric 的作者正在合并一些和 Python 3 相关的改动。如果完成修改，那下面的例子可以正常运行。不过，在那之前还是需要使用 Python 2 来运行。

首先，用下面的命令安装 fabric：

```
$ pip2 install fabric
```

可以不使用 SSH，直接用 fabric 运行本地 Python 代码。把下面的代码保存为 fab1.py：

```
def iso():
    from datetime import date
    print(date.today().isoformat())
```

接着，输入下面的命令来运行：

```
$ fab -f fab1.py -H localhost iso

[localhost] Executing task 'iso'
2014-02-22

Done.
```

-f fab1.py 选项指定使用 fabric 文件 fab1.py，而不是默认的 fabfile.py。-H localhost 选项指定运行本地的命令。最后，iso 是 fab 文件中要运行的函数名。它的工作原理和之前的 RPC 有点像。具体的选项参见官方文档（http://docs.fabfile.org/）。

要在本地或者远程运行外部程序，机器必须运行 SSH 服务器。在 Unix 类系统中，服务器是 sshd；service sshd status 可以检查服务器是否启动，如果需要，可以使用 service sshd start 来启动它。在 Mac 中，打开“系统偏好设置”，点击“共享”，然后勾选“远程

登录”。Windows 没有内置的 SSH 支持，建议安装 putty（http://www.chiark.greenend.org.uk/~sgtatham/putty/）。

我们还是使用了函数名 `iso`，但这次使用 `local()` 来运行命令。下面是代码和输出：

```
from fabric.api import local

def iso():
    local('date -u')
```

```
$ fab -f fab2.py -H localhost iso

[localhost] Executing task 'iso'
[localhost] local: date -u
Sun Feb 23 05:22:33 UTC 2014

Done.
Disconnecting from localhost... done.
```

`local()` 对应的远程方法是 `run()`。下面是 fab3.py：

```
from fabric.api import run

def iso():
    run('date -u')
```

`fabric` 在遇到 `run()` 时会使用 SSH 连接命令行中用 `-H` 指定的主机。如果你有本地网络并且可以使用 SSH 连接一个主机，那可以在 `-H` 之后加上那个主机名（就像下面的示例一样）。如果没有这样的主机，那就使用 `localhost`，`fabric` 会像访问远程机器一样访问它，这在测试时很有用。本例还是使用 `localhost`：

```
$ fab -f fab3.py -H localhost iso

[localhost] Executing task 'iso'
[localhost] run: date -u
[localhost] Login password for 'yourname':
[localhost] out: Sun Feb 23 05:26:05 UTC 2014
[localhost] out:


Done.
Disconnecting from localhost... done.
```

注意，我需要输入密码来登录。如果想省略这一步，可以在 fabric 文件中写入密码：

```
from fabric.api import run
from fabric.context_managers import env

env.password = "your password goes here"

def iso():
    run('date -u')
```

运行它：

```
$ fab -f fab4.py -H localhost iso

[localhost] Executing task 'iso'
[localhost] run: date -u
[localhost] out: Sun Feb 23 05:31:00 UTC 2014
[localhost] out:


Done.
Disconnecting from localhost... done.
```

把密码放在代码中非常不安全。更好的方法是使用公钥和密钥配置 SSH，可以使用 `ssh-keygen`（https://help.github.com/articles/generating-ssh-keys/）。

3. Salt

Salt（http://saltstack.com/）最初的目的是实现远程运行，但是后来变成了一个完整的系统管理平台。它是基于 ZeroMQ 开发的，不是基于 SSH，因此可以扩展到上千台服务器。

Salt 还没有兼容 Python 3，这里我不会提供 Python 2 的示例代码。如果你对它感兴趣，可以阅读文档并等待它兼容 Python 3。

类似的产品有 `puppet`（http://puppetlabs.com/）和 `chef`（http://www.getchef.com/chef/），它们和 Ruby 关系密切。`ansible`（http://www.ansible.com/home）包是 Python 写成的另一个类似 Salt 的系统，也值得一试。它可以免费下载和使用，但是支持和一些插件包需要商业许可。它默认使用 SSH，并且并不需要在机器上安装其他特殊软件。

`salt` 和 `ansible` 都包含了 `fabric` 的功能，可以进行初始化配置、部署和远程执行。

11.2.10　大数据和MapReduce

当 Google 和其他互联网公司成长起来之后，它们发现传统的计算机解决方案不能扩展。可以运行在单机或者少量机器上的软件无法支持上千台机器。

存储数据的数据库和文件需要多次寻道，这会产生多次磁头移动。（想想黑胶唱片和它移动唱针的时间，再想想唱针放下时造成的噪音和人们说话的声音。）但是，连续读取磁盘上的区块时速度很快。

开发者发现把数据分布在网络的不同机器上并进行分析会比只用一台机器快很多。它们会使用那些听起来很简单但是效率很高的算法来快速处理分布式数据。其中之一就是 MapReduce，它可以在许多机器上执行计算并收集结果，很像队列。

Google 在论文中发表这个成果之后，Yahoo 发布了一个基于 Java 的开源包，名为 Hadoop（这个名字来源于项目领导者儿子的一个玩具大象）。

这里要说一下*大数据*这个词。通常来说，它的意思是“数据对于我的机器来说太大了”：数据超出了已有的磁盘、内存、CPU 时间或者所有这些。对于某些组织来说，一旦遇到大数据问题，那解决方案总是 Hadoop。Hadoop 会把数据复制到其他机器上，通过 map 和 reduce 程序来处理它们并把每一步的结果存储到磁盘上。

这个过程可能很慢。更快的方法是 Hadoop *流*，就像 Unix 的管道一样，把每一步产生的数据流直接传输给下一步，这样就可以避免存储到磁盘。你可以用任何语言来编写 Hadoop 流程序，包括 Python。

已经有很多关于 Hadoop 的 Python 模块，“Python Hadoop 框架教程”（http://blog.cloudera.com/blog/2013/01/a-guide-to-python-frameworks-for-hadoop/）这篇博文介绍了很多。Spotify 公司的流媒体音乐很出名，它开源了自己处理 Hadoop 流的 Python 部件 Luigi（https://github.com/spotify/luigi）。不过现在还不兼容 Python 3。

Hadoop 有一个竞争对手 Spark（http://spark.apache.org/docs/latest/index.html），它的目标是大大加快运行速度。它可以读取和处理所有 Hadoop 的数据结构和格式。Spark 包含 Python 和其他语言的 API，可以参见在线安装文档（http://spark.apache.org/downloads.html）。

另一个类似的产品是 Disco（http://discoproject.org/），它使用 Python 来完成 MapReduce 过程，使用 Erlang 完成通信部分。不过，只可惜无法使用 `pip` 来安装，具体方法参见文档（http://disco.readthedocs.org/en/latest/start/download.html）。

附录 C 有*并行编程*相关的示例，可以在分布式集群中执行大规模结构化计算。

11.2.11　在云上工作

不久之前，你还需要买自己的服务器，把它们放在数据中心的机柜上，安装各种软件：操作系统、设备驱动、文件系统、数据库、Web 服务器、邮件服务器、域名服务器、负载均衡、监控程序，等等。当你做过很多遍之后，就会失去新鲜感，并且需要一直担心安全问题。

许多托管服务都提供有偿维护，但是你仍然需要租用物理设备并且按照峰值负载来付费。

机器数量多了之后，就很容易出现问题。你需要横向扩展服务并对数据做冗余存储。网络操作和单机完全不同，Peter Deutsch 说过，分布式计算的八大误解是：

- 网络是可靠的；
- 延迟为零；
- 带宽无限；
- 网络是安全的；
- 拓扑结构不会改变；
- 传输成本为零；
- 网络是同构的。

你可以试着搭建复杂的分布式系统，但这非常困难，并且需要另一组工具集。借用一个比喻，如果你只有少数几个服务器，你会像对待宠物一样对待它们——给它们命名，了解它们的特点，在需要时尽量治疗它们。但是规模变大之后，你像对待牲口一样对待它们：它

们看起来都一样，每个都有编号，如果遇到问题可以被替换掉。

除了自己搭建，你还可以租用云上的服务器。使用这种模式时，维护是其他人的问题，你可以专注在你的服务、博客或者任何你想展示给世界的东西上。使用 Web 仪表盘和 API 可以快速和轻松地创建任何你需要的服务器——它们是有弹性的。你可以监控它们的状态，如果某些参数超过阈值会收到提醒。目前，云是一个非常火的话题，企业在云组件上的支出在不断飙升。

下面我们来看看如何在 Python 中使用现在流行的云平台。

1. Google

Google 内部大量使用 Python，它还招聘了很多高级 Python 开发者（连吉多 · 范 · 罗苏姆都工作过一段时间）。

打开 App Engine 网站（https://developers.google.com/appengine/），在“选择语言”下面点击 Python。你可以在云编辑器中输入 Python 代码，可以直接在下方看到运行结果。在结果后面是链接，可以下载 Python SDK，这样你就可以在自己的硬件上使用 Google 的云 API 进行开发。下面是把应用部署到 AppEngine 的一些细节。

在 Google 云的主页（https://cloud.google.com/）上可以找到服务的详细介绍。

- App Engine
  一个高层平台，包含一些 Python 工具，比如 `flask` 和 `django`。
- Compute Engine
  创建一个虚拟机集群来进行大规模分布式计算。
- Cloud Storage
  对象存储（对象是文件，但是没有目录结构）。
- Cloud Datastore
  大型 NoSQL 数据库。
- Cloud SQL
  大型 SQL 数据库。
- Cloud Endpoints
  用 Restful 来访问应用。
- BigQuery
  类似 Hadoop 的大数据处理。

如果硬要说，Google 的服务在和 Amazon、OpenStack 竞争。

2. Amazon

当 Amazon 的服务器数量剧增之后，开发者遇到了许多分布式系统带来的问题。大约是 2002 年的某一天，CEO Jeff Bezos 向所有员工宣布，从今往后，Amazon 的所有数据和功能都要通过网络服务接口来使用——再也没有文件、数据库或者本地调用。他们必须把这些接口设计成可以公开使用。最后 Jeff 说：“做不到的人会被解雇。”

不出所料，开发者们开发出一个非常大的面向服务的架构。他们借鉴了很多解决方案，最终完成了 Amazon Web Services（AWS，http://aws.amazon.com/cn/）。这个庞然大物现在已经统治了市场。目前，AWS 包含很多服务。和我们关系最密切的有以下这些服务。

- Elastic Beanstalk
  高层应用平台
- EC2（Elastic Compute）
  分布式计算
- S3（Simple Storage Service）
  对象存储
- RDS
  关系数据库（MySQL、PostgreSQL、Oracle、MSSQL）
- DynamoDB
  NoSQL 数据库
- Redshift
  数据仓库
- EMR
  Hadoop

更多关于 AWS 服务的细节，请下载 Amazon Python SDK（http://aws.amazon.com/developers/getting-started/python/）并阅读帮助部分。

官方的 Python AWS 库 `boto`（http://docs.pythonboto.org/en/latest/）是另一个类似的工具，还没有完全兼容 Python 3。你需要使用 Python 2 或者使用其他类似的工具，可以在 Python 包索引（https://pypi.python.org/pypi）中搜索“aws”或者“amazon”。

3. OpenStack

第二个非常流行的云服务是由 Rackspace 提供的。2010 年，Rackspace 和 NASA 达成了不寻常的合作关系，把它们的一些云设施合并成了 OpenStack（http://www.openstack.org/）。这是一个免费的开源平台，可以搭建公有云、私有云和混合云。每 6 个月发布一个新版本，最近的版本有超过 125 万 Python 代码，有很多贡献者。越来越多的组织在产品中使用 OpenStack，包括 CERN 和 PayPal。

OpenStack 的主要 API 是 RESTful 的，它的 Python 模块还提供了程序级别的接口和用于 shell 自动化的命令行 Python 程序。下面是当前发行版中的一些标准访问。

- Keystone
  认证服务，提供认证（比如用户名 / 密码）、授权（功能）和服务发现。
- Nova
  计算服务，通过网络上的服务器进行分布式工作。

- Swift
  对象存储，类似 Amazon 的 S3。它被用于 Rackspace 的 Cloud Files 服务。
- Glance
  中层的镜像存储服务。
- Cinder
  低层次的块存储服务。
- Horizon
  基于 Web 的所有服务的仪表盘。
- Neutron
  网络管理服务。
- Heat
  配置管理（多个云）服务。
- Ceilometer
  遥测（度量、监控）服务。

经常有新服务被提出，有些经过孵化过程后，可能会成为标准 OpenStack 平台的一部分。

OpenStack 运行在 Linux 或者 Linux 虚拟机中。核心服务的安装还是有些复杂。在 Linux 上安装 OpenStack 最快捷的方法就是使用 Devstack（http://docs.openstack.org/developer/devstack/）来一键安装。完成后，你可以使用一个 Web 仪表盘来查看和配置其他服务。

如果想手动安装 OpenStack，可以使用 Linux 的包管理工具。所有的主要 Linux 发行版都支持 OpenStack 并且在下载服务器上提供了官方安装包。可以在 OpenStack 官网上查看安装文档、新闻和相关信息。

OpenStack 的开发和企业支持正在逐步推进，很像当年 Linux 和 Unix 竞争的情景。

11.3 练习

(1) 使用原始的 `socket` 来实现一个获取当前时间的服务。当客户端向服务器发送字符串 time 时，服务器会返回当前日期和时间的 ISO 格式字符串。

(2) 使用 ZeroMQ 的 REQ 和 REP 套接字实现同样的功能。

(3) 使用 XMLRPC 实现同样的功能。

(4) 你可能看过那部很老的《我爱露西》（*I Love Lucy*）电视节目。露西和埃塞尔在一个巧克力工厂里工作（这是传统）。他们落在了运输甜点的传送带后面，所以必须用更快的速度进行处理。写一个程序来模拟这个过程，程序会把不同类型的巧克力添加到一个 Redis 列表中，露西是一个客户端，对列表执行阻塞的弹出操作。她需要 0.5 秒来处理一块巧克力。打印出时间和露西处理的每块巧克力类型以及剩余巧克力的数量。

(5) 使用 ZeroMQ 发布第 7 章练习 (7) 中的诗（参见 7.3 节），每次发布一个单词。写一个 ZeroMQ 客户端来打印出每个以元音开头的单词，再写另一个客户端来打印出所有长度为 5 的单词。忽略标点符号。

第 12 章

成为真正的Python开发者

“想回到过去揍年轻的自己吗？选择软件开发作为你的事业吧！”

—— Elliot Loh（https://twitter.com/loh/status/411282297816498176）

本章会介绍 Python 开发中的艺术和科学，还有一些“最佳实践”。掌握它们之后，你就可以成为一个正牌的 Python 开发者。

12.1　关于编程

首先是一些基于我个人经历的和编程相关的建议。

最初我准备走科学这条路，我自学了编程来分析和展示实验数据。我以为计算机编程和会计一样——准确但是无聊。当我发现自己真正享受这个过程时非常惊讶。部分乐趣来自逻辑方面，就像解开谜题一样，但是还有一部分是创造力。你必须正确地编写程序才能得到正确的结果，但是可以用任何喜欢的方式来完成它。这是一种不寻常的左右脑平衡思考训练。

在掉进编程这个大坑后，我发现这个领域有很多方向，每个方向都有很多不同的任务和不同的人。你可以选择计算机图形学、操作系统、商业应用，甚至科学。

如果你是一个程序员，可能也有类似的经历。如果你不是，或许应该尝试一下编程，看看是否符合你的个性，至少也可以帮你完成一些工作。就像我在本书前面提到的一样，数学能力并不是那么重要。看起来逻辑思考的能力最重要，语言能力也很有用。最后，耐心很重要，尤其是寻找代码中的 bug 时。

12.2　寻找Python代码

当你需要开发功能时，最快的解决方法就是“偷”。好吧……是从一个经过允许的来源

“偷”代码。

Python 标准库（http://docs.python.org/3/library/）既宽又深，而且特别整洁。深入进去能学到很多东西。

就像各种体育运动的名人堂一样，一个模块需要经过时间的检验才能加入标准库。新的包一直在出现，本书已经介绍过一些，它们有的能做一些新的东西，有些可以更好地完成旧的工作。Python 的广告语是内置电池（batteries included），但你可能需要一种新电池。

所以，除了标准库之外，还能去哪儿寻找优秀的 Python 代码呢？

首先要推荐的就是 Python 包索引（PyPi，https://pypi.python.org/pypi）。之前被命名为巨蟒剧团中的奶酪店（Cheese Shop）。这个网站上的 Python 包一直在更新，我编写本书时已经超过了 39 000 个。当你使用 `pip`（参见下一节）时它就会搜索 PyPi。PyPi 的主页显示的是最近添加的包。你也可以直接搜索。举例来说，表 12-1 列出了 `genealogy` 的搜索结果。

表12-1：PyPi中搜索genealogy的结果

包	权重	描述
Gramps 3.4.2	5	分析、组织并分享你的家谱
python-fs-stack 0.2	2	Python 封装过的所有 FamilySearch API
human-names 0.1.1	1	人名
nameparser 0.2.8	1	一个简单的 Python 模块，用于将人名解析成具体的组成部件

最匹配的包权重最高，因此 `Gramps` 看起来最符合要求。可以去 Python 网站（https://pypi.python.org/pypi/Gramps/3.4.2）来查看文档和下载链接。

另一个很流行的仓库是 GitHub。可以在“流行”（https://github.com/trending?l=python）中查看当前流行的 Python 包。

流行 Python 菜谱（http://code.activestate.com/recipes/langs/python/）有 4000 多个短 Python 程序，涉及多个方面。

12.3　安装包

有 3 种安装 Python 包的方法：

- 推荐使用 `pip`，你可以使用 `pip` 来安装绝大多数 Python 包；
- 有时可以使用操作系统自带的包管理工具；
- 从源代码安装。

如果你对同一个领域的多个包感兴趣，或许可以找到一个包含这些包的 Python 发行版。举例来说，在附录 C 中，可以使用一系列数学和科学程序，如果手动安装很麻烦，可以使用类似 Anaconda 这样的发行版。

12.3.1　使用pip

Python 的包有一些限制。之前有一个安装工具叫 `easy_install`，现在已经被 `pip` 替代了，

但是它们都没有成为标准的 Python 安装工具。如果想使用 `pip`，如何安装它呢？从 Python 3.4 开始，`pip` 终于成为了 Python 的一部分，避免了不必要的步骤。如果使用的是 Python 3 之前的版本并且没有 `pip`，那你可以从 http://www.pip-installer.org 下载。

`pip` 最简单的使用方法就是通过下面的命令安装一个包的最新版：

```
$ pip install flask
```

你会看到详细的安装过程，这样就可以确保安装正常进行：下载，运行 `setup.py`，在硬盘上安装文件，等等。

也可以要求 `pip` 安装指定的版本：

```
$ pip install flask==0.9.0
```

或者指定最小版本（当你必须使用的一些特性在某个版本之后开始出现时，这个功能特别有用）：

```
$ pip install 'flask>=0.9.0'
```

在这条命令中，单引号可以防止 shell 把 `>` 解析成输出重定向，那样会把输出写入一个名为 `=0.9.0` 的文件中。

如果你想安装多个 Python 包，可以使用 requirements 文件（https://pip.pypa.io/en/latest/reference/pip_install.html#requirements-file-format）。虽然它有很多选项，但是最简单的使用方法是列出所有包，一个包一行，加上可选的目标版本或者相对版本：

```
$ pip -r requirements.txt
```

你的示例 requirements.txt 文件可能是这样：

```
flask==0.9.0
django
psycopg2
```

12.3.2 使用包管理工具

苹果的 OS X 中有第三方包管理工具 homebrew（`brew`，http://brew.sh/）和 `ports`（http://www.macports.org/）。它们的原理和 `pip` 类似，但并不是只能安装 Python 包。

Linux 的不同发行版有不同的包管理工具，最流行的是 `apt-get`、`yum`、`dpkg` 和 `zypper`。

Windows 有 Windows 安装工具，需要后缀为 `.msi` 的包文件。如果想在 Windows 上安装 Python，那可能就是 MSI 格式的。

12.3.3 从源代码安装

有时候，一个 Python 包是新出的，或者作者还没有把它发布到 `pip` 上。如果要安装这样的包，通常需要这样做：

(1) 下载代码；

(2) 如果是压缩文件，使用 `zip`、tar 或者其他合适的工具来解压缩；
(3) 在包含 setup.py 文件的目录中运行 `python install setup.py`。

> 下载和安装时一定要小心。虽然在 Python 程序中加入病毒难度不小（因为这些程序是可读的文本），有时还是会遇到有毒的程序。

12.4 集成开发环境

我编写本书中的软件时使用的是纯文本编辑器，但是你不一定非要在命令行窗口或者纯文本编辑器中写代码。有许多免费和收费的集成开发环境（IDE），它们有图形窗口（GUI）并且支持文本编辑器、调试器、库搜索，等等。

12.4.1 IDLE

IDLE（https://docs.python.org/3/library/idle.html）是唯一一个标准发行版中包含的 Python IDE。它是基于 tkinter 开发的，图形界面比较简单。

12.4.2 PyCharm

PyCharm（http://www.jetbrains.com/pycharm/）是一个新出的 IDE，有许多特性。社区版是免费的，你也可以使用学生身份或者开源项目来获取免费的专业版许可。图 12-1 是初始界面。



图 12-1：PyCharm 的初始界面

12.4.3　IPython

IPython（http://ipython.org/）在附录 C 中有介绍，既是一个发布平台也是一个扩展 IDE。

12.5　命名和文档

你绝对记不住自己写过什么。很多次我看着自己最近写的代码，心里想的是：它们到底是从哪儿来的。这就是文档的重要性。文档可以包括注释和文档字符串，也可以把信息记录在变量名、函数名、模块名和类名中。不要像下面这样啰嗦：

```
>>> # 这里我要给变量"num"赋值10：
... num = 10
>>> # 我希望它确实赋值成功了
... print(num)
10
>>> # 好的
```

相反，要说清楚为什么要赋值 `10`，为什么变量名是 `num`。如果在编写华氏到摄氏度的转换，那应该让变量名自己表达它们的意义，而不是写一些魔法代码。加一些测试更好：

```
def ftoc(f_temp):
    "把华氏温度<f_temp>转换为摄氏温度并返回"
    f_boil_temp = 212.0
    f_freeze_temp = 32.0
    c_boil_temp = 100.0
    c_freeze_temp = 0.0
    f_range = f_boil_temp - f_freeze_temp
    c_range = c_boil_temp - c_freeze_temp
    f_c_ratio = c_range / f_range
    c_temp = (f_temp - f_freeze_temp) * f_c_ratio + c_freeze_temp
    return c_temp

if __name__ == '__main__':
    for f_temp in [-40.0, 0.0, 32.0, 100.0, 212.0]:
        c_temp = ftoc(f_temp)
        print('%f F => %f C' % (f_temp, c_temp))
```

运行测试：

```
$ python ftoc1.py

-40.000000 F => -40.000000 C
0.000000 F => -17.777778 C
32.000000 F => 0.000000 C
100.000000 F => 37.777778 C
212.000000 F => 100.000000 C
```

这段代码（至少）有两处可以改进的地方。

- Python 没有常量，但是 PEP8 格式规范建议（http://legacy.python.org/dev/peps/pep-0008/#constants）使用大写字母和下划线（比如 `ALL_CAPS`）来表示常量名。我们据此修改示例中的常量。

• 我们基于常量值提前进行了一些计算，应该把它们移动到模块顶层，这样它们就只会计算一次，否则每次调用 ftoc() 时都需要计算。

修改后的版本如下所示：

```
F_BOIL_TEMP = 212.0
F_FREEZE_TEMP = 32.0
C_BOIL_TEMP = 100.0
C_FREEZE_TEMP = 0.0
F_RANGE = F_BOIL_TEMP - F_FREEZE_TEMP
C_RANGE = C_BOIL_TEMP - C_FREEZE_TEMP
F_C_RATIO = C_RANGE / F_RANGE

def ftoc(f_temp):
    "把华氏温度<f_temp>转换为摄氏温度并返回"
    c_temp = (f_temp - F_FREEZE_TEMP) * F_C_RATIO + C_FREEZE_TEMP
    return c_temp

if __name__ == '__main__':
    for f_temp in [-40.0, 0.0, 32.0, 100.0, 212.0]:
        c_temp = ftoc(f_temp)
        print('%f F => %f C' % (f_temp, c_temp))
```

12.6 测试代码

有时候，我会修改一些代码，然后对自己说："看起来没问题，发布吧。"接着程序就出问题了。唉，每次遇到这种情况时（还好这种情况已经越来越少了），我都觉得自己是个笨蛋并发誓下次要写更多的测试。

测试 Python 程序最简单的办法就是添加一些 print() 语句。Python 交互式解释器的读取 – 求值 – 打印循环（REPL）允许快速添加和测试修改。然而，你或许不会想要在产品级代码中添加 print() 语句，因此需要记住自己添加的所有 print() 语句并在最后删除它们。但是，这样做很容易出现剪切 – 粘贴错误。

12.6.1 使用pylint、pyflakes和pep8检查代码

在创建真实的测试程序之前，需要运行 Python 代码检查器。最流行的是 pylint（http://www.pylint.org/）和 pyflakes（https://pypi.python.org/pypi/pyflakes/）。你可以使用 pip 来安装它们：

```
$ pip install pylint
$ pip install pyflakes
```

它们可以检查代码错误（比如在赋值之前引用变量）和代码风格问题（就像穿着格子衣服和条纹衣服一样）。下面是一段毫无意义的程序，有一个 bug 和一个风格问题：

```
a = 1
b = 2
print(a)
print(b)
print(c)
```

下面是 pylint 输出内容的一部分：

```
$ pylint style1.py

No config file found, using default configuration
************* Module style1
C:  1,0: Missing docstring
C:  1,0: Invalid name "a" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  2,0: Invalid name "b" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
E:  5,6: Undefined variable 'c'
```

往下翻，Global evaluation 下面是我们的分数（最高为 10.0）：

```
Your code has been rated at -3.33/10
```

好吧。我们首先来修复 bug。pylint 输出中以 E 开头的表示这是一个 Error（错误），原因是我们在给 c 赋值之前打印了它。修复一下：

```
a = 1
b = 2
c = 3
print(a)
print(b)
print(c)

$ pylint style2.py

No config file found, using default configuration
************* Module style2
C:  1,0: Missing docstring
C:  1,0: Invalid name "a" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  2,0: Invalid name "b" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
C:  3,0: Invalid name "c" for type constant
  (should match (([A-Z_][A-Z0-9_]*)|(__.*__))$)
```

好的，没有 E 了，分数也从 −3.33 变成了 4.29：

```
Your code has been rated at 4.29/10
```

pylint 需要一个文档字符串（出现在模块或者函数内部第一行的一段短文本，用来描述代码），而且它认为短变量名 a、b 和 c 太土了。我们来让 pylint 高兴点儿，把 style2.py 修改为 style3.py：

```
"这里是模块文档字符串"

def func():
    "函数的文档字符串在这里。妈妈我在这儿！"
    first = 1
    second = 2
    third = 3
    print(first)
```

```
    print(second)
    print(third)

func()
```

```
$ pylint style3.py
```

```
No config file found, using default configuration
```

嘿，没有任何抱怨了。我们的分数呢？

```
Your code has been rated at 10.00/10
```

还可以，是吧？

另一个格式检查工具是 pep8（https://pypi.python.org/pypi/pep8），可以使用熟悉的方式安装：

```
$ pip install pep8
```

它对我们的代码有何意见呢？

```
$ pep8 style3.py
```

```
style3.py:3:1: E302 expected 2 blank lines, found 1
```

为了满足格式要求，它建议我在文档字符串后面添加一个空行。

12.6.2　使用unittest进行测试

我们已经通过了代码风格的考验，下面该真正地测试程序逻辑了。

最好先编写独立的测试程序，在提交代码到源码控制系统之前确保通过所有测试。写测试看起来是一件很麻烦的事，但是它们真的能帮助你更快地发现问题，尤其是*回归测试*（破坏之前还能正常工作的代码）。工程师们已经从惨痛的经历中领悟到一个真理：即使是很小的看起来没有任何问题的改动，也可能出问题。如果看那些优秀的 Python 包就会发现，它们大多都有测试集。

标准库中有两个测试包。首先介绍 unittest。假设我们编写了一个单词首字母转大写的模块，第一版直接使用标准字符串函数 capitalize()，之后会看到许多意料之外的结果。把下面的代码保存为 cap.py：

```
def just_do_it(text):
    return text.capitalize()
```

测试就是先确定输入对应的期望输出（本例期望的输出是输入文本的首字母大写版本），然后把输入传入需要测试的函数，并检查返回值和期望输出是否相同。期望输出被称为*断言*，因此在 unittest 中，可以使用 assert（断言）开头的方法来检查返回的结果，比如下面代码中的 assertEqual 方法。

把下面的测试脚本保存为 test_cap.py：

```
import unittest
import cap
```

```
class TestCap(unittest.TestCase):

    def setUp(self):
        pass

    def tearDown(self):
        pass

    def test_one_word(self):
        text = 'duck'
        result = cap.just_do_it(text)
        self.assertEqual(result, 'Duck')

    def test_multiple_words(self):
        text = 'a veritable flock of ducks'
        result = cap.just_do_it(text)
        self.assertEqual(result, 'A Veritable Flock Of Ducks')


if __name__ == '__main__':
    unittest.main()
```

setUp() 方法会在每个测试方法执行之前执行，tearDown() 方法是在每个测试方法执行之后执行。它们通常用来分配和回收测试需要的外部资源，比如数据库连接或者一些测试数据。在本例中，我们的测试方法已经足够进行测试，因此不需要再定义 setUp() 和 tearDown()，但是放一个空方法也没关系。我们测试的核心是函数 test_one_word() 和 test_multiple_words()。它们会运行我们定义的 just_do_it() 函数，传入不同的输出并检查返回值是否和期望输出一样。

运行一下这个脚本，它会调用那两个测试方法：

```
$ python test_cap.py

 F.
 ======================================================================
 FAIL: test_multiple_words (__main__.TestCap)
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "test_cap.py", line 20, in test_multiple_words
  self.assertEqual(result, 'A Veritable Flock Of Ducks')
 AssertionError: 'A veritable flock of ducks' != 'A Veritable Flock Of Ducks'
 - A veritable flock of ducks
 ?   ^         ^     ^  ^
 + A Veritable Flock Of Ducks
 ?   ^         ^     ^  ^


 ----------------------------------------------------------------------
 Ran 2 tests in 0.001s

 FAILED (failures=1)
```

看起来第一个测试（test_one_word）通过了，但是第二个（test_multiple_words）失败

了。上箭头（^）指出了字符串不相同的地方。

为什么多个单词会失败？可以阅读 string 的 capitalize（https://docs.python.org/3/library/stdtypes.html#str.capitalize）函数文档来寻找线索：它只会把第一个单词的第一个字母转成大写。或许我们应该先阅读文档。

为了修复错误，我们需要另一个函数。往下翻一翻网页，可以看到 title() 函数（https://docs.python.org/3/library/stdtypes.html#str.title）。我们把 cap.py 中的 capitalize() 替换成 title()：

```
def just_do_it(text):
    return text.title()
```

再次运行测试，看看结果如何：

```
$ python test_cap.py

 ..
 ----------------------------------------------------------------------
 Ran 2 tests in 0.000s

 OK
```

看起来没问题了。不过，其实还是有问题的。我们还需要在 test_cap.py 中添加另一个方法：

```
    def test_words_with_apostrophes(self):
        text = "I'm fresh out of ideas"
        result = cap.just_do_it(text)
        self.assertEqual(result, "I'm Fresh Out Of Ideas")
```

再试一次：

```
$ python test_cap.py

 ..F
 ======================================================================
 FAIL: test_words_with_apostrophes (__main__.TestCap)
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "test_cap.py", line 25, in test_words_with_apostrophes
     self.assertEqual(result, "I'm Fresh Out Of Ideas")
 AssertionError: "I'M Fresh Out Of Ideas" != "I'm Fresh Out Of Ideas"
 - I'M Fresh Out Of Ideas
 ?   ^
 + I'm Fresh Out Of Ideas
 ?   ^


 ----------------------------------------------------------------------
 Ran 3 tests in 0.001s

 FAILED (failures=1)
```

函数把 I'm 中的 m 大写了。浏览一下 title() 的文档会发现，它不能处理撇号。我们真得应该先完整地阅读一遍文档。

在标准库 string 文档的底部有另一个函数：一个名为 capwords() 的辅助函数。试试这个：

```
def just_do_it(text):
    from string import capwords
    return capwords(text)
```

```
$ python test_cap.py

 ...
 ----------------------------------------------------------------------
 Ran 3 tests in 0.004s

 OK
```

终于完成了！呃，还有问题。向 test_cap.py 中再加一个测试：

```
def test_words_with_quotes(self):
    text = "\"You're despicable,\" said Daffy Duck"
    result = cap.just_do_it(text)
    self.assertEqual(result, "\"You're Despicable,\" Said Daffy Duck")
```

能通过吗？

```
$ python test_cap.py

 ...F
 ======================================================================
 FAIL: test_words_with_quotes (__main__.TestCap)
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "test_cap.py", line 30, in test_words_with_quotes
     self.assertEqual(result, "\"You're
     Despicable,\" Said Daffy Duck")
 AssertionError: '"you\'re Despicable," Said Daffy Duck'
  != '"You\'re Despicable," Said Daffy Duck'
 - "you're Despicable," Said Daffy Duck
 ?  ^
 + "You're Despicable," Said Daffy Duck
 ?  ^


 ----------------------------------------------------------------------
 Ran 4 tests in 0.004s

 FAILED (failures=1)
```

似乎第一个双引号并没有被目前为止最好用的函数 capwords 正确处理。它试着把 " 转成大写，并把其他内容转成小写（You're）。此外，字符串的其余部分应该保持不变。

做测试的人可以发现这些边界条件，但是开发者在面对自己的代码时通常有盲区。

unittest 提供了数量不多但非常有用的断言，你可以用它们检查值、确保类能够匹配、判断是否触发错误，等等。

12.6.3 使用doctest进行测试

标准库中的第二个测试包是 `doctest`（https://docs.python.org/3/library/doctest.html）。使用这个包可以把测试写到文档字符串中，也可以起到文档的作用。它看起来有点像交互式解释器：字符 >>> 后面是一个函数调用，下一行是执行结果。你可以在交互式解释器中运行测试并把结果粘贴到测试文件中。我们修改一下 cap.py（暂时不考虑那个双引号的问题）：

```
def just_do_it(text):
    """
    >>> just_do_it('duck')
    'Duck'
    >>> just_do_it('a veritable flock of ducks')
    'A Veritable Flock Of Ducks'
    >>> just_do_it("I'm fresh out of ideas")
    "I'm Fresh Out Of Ideas"
    """
    from string import capwords
    return capwords(text)

if __name__ == '__main__':
    import doctest
    doctest.testmod()
```

运行时如果测试全部通过不会产生任何输出：

```
$ python cap.py
```

加上冗杂选项（-v），看看会出现什么：

```
$ python cap.py -v

 Trying:
     just_do_it('duck')
 Expecting:
     'Duck'
 ok
 Trying:
     just_do_it('a veritable flock of ducks')
 Expecting:
     'A Veritable Flock Of Ducks'
 ok
 Trying:
     just_do_it("I'm fresh out of ideas")
 Expecting:
     "I'm Fresh Out Of Ideas"
 ok
 1 items had no tests:
     __main__
 1 items passed all tests:
    3 tests in __main__.just_do_it
 3 tests in 2 items.
 3 passed and 0 failed.
 Test passed.
```

12.6.4　使用nose进行测试

第三方包 nose（https://nose.readthedocs.org/en/latest/）和 unittest 类似。下面是安装命令：

```
$ pip install nose
```

不需要像使用 unittest 一样创建一个包含测试方法的类。任何名称中带 test 的函数都会被执行。我们修改一下之前的 unittest 示例并保存为 test_cap_nose.py：

```
import cap
from nose.tools import eq_

def test_one_word():
    text = 'duck'
    result = cap.just_do_it(text)
    eq_(result, 'Duck')

def test_multiple_words():
    text = 'a veritable flock of ducks'
    result = cap.just_do_it(text)
    eq_(result, 'A Veritable Flock Of Ducks')

def test_words_with_apostrophes():
    text = "I'm fresh out of ideas"
    result = cap.just_do_it(text)
    eq_(result, "I'm Fresh Out Of Ideas")

def test_words_with_quotes():
    text = "\"You're despicable,\" said Daffy Duck"
    result = cap.just_do_it(text)
    eq_(result, "\"You're Despicable,\" Said Daffy Duck")
```

运行测试：

```
$ nosetests test_cap_nose.py

 ...F
 ======================================================================
 FAIL: test_cap_nose.test_words_with_quotes
 ----------------------------------------------------------------------
 Traceback (most recent call last):
   File "/Users/.../site-packages/nose/case.py", line 198, in runTest
     self.test(*self.arg)
   File "/Users/.../book/test_cap_nose.py", line 23, in test_words_with_quotes
     eq_(result, "\"You're Despicable,\" Said Daffy Duck")
 AssertionError: '"you\'re Despicable," Said Daffy Duck'
     != '"You\'re Despicable," Said Daffy Duck'

 ----------------------------------------------------------------------
 Ran 4 tests in 0.005s

 FAILED (failures=1)
```

这和我们使用 unittest 进行测试得到的错误一样。幸运的是，在本章结尾的练习中，我们会修复它。

12.6.5　其他测试框架

出于某些原因，开发者会编写 Python 测试框架。如果你很感兴趣，可以试试 `tox`（http://tox.readthedocs.org/en/latest/）和 `py.test`（http://pytest.org/latest/）。

12.6.6　持续集成

当你的团队每天会产生很多代码时，就需要在出现改动时进行自动测试。你可以让源码控制系统，在提交代码时进行自动化测试。这样每个人都知道是谁破坏了构建并消失在吃午饭的人群中。

这些系统很庞大，我不会在这里介绍安装和使用方法。如果某一天，你需要用到它们，至少知道可以去哪儿找。

- `buildbot`（http://buildbot.net/）

  用 Python 写成的源码控制系统，可以自动构建、测试和发布。

- `jenkins`（http://jenkins-ci.org/）

  用 Java 写成，应该是目前最受欢迎的 CI（持续集成）系统。

- `travis-ci`（http://travis-ci.com/）

  这个自动化项目托管在 GitHub 上，对开源项目是免费的。

12.7　调试Python代码

> “调试的难度是写代码的两倍。以此类推，如果你绞尽脑汁编写巧妙的代码，那你一定无法调试它。”
>
> ——Brian Kernighan

测试先行。测试越完善，之后需要修复的 bug 越少。不过，bug 总是无法避免的，发现 bug 就要去修复它。之前就说过，Python 中最简单的调试方法就是打印字符串。`vars()` 是非常有用的一个函数，可以提取本地变量的值，包括函数参数：

```
>>> def func(*args, **kwargs):
...     print(vars())
...
>>> func(1, 2, 3)
{'args': (1, 2, 3), 'kwargs': {}}
>>> func(['a', 'b', 'argh'])
{'args': (['a', 'b', 'argh'],), 'kwargs': {}}
```

就像 4.9 节介绍的，装饰器可以在不修改函数代码的前提下，在函数运行之前或者之后执行其他代码。这意味着你可以使用装饰器在任何 Python 函数（不仅是你自己写的函数）之前或者之后做一些事情。我们定义一个装饰器 `dump`，它可以打印出输入的参数和函数的返回值（对设计师来说，垃圾堆通常需要装饰一下[1]）：

注 1：dump 的意思是倾倒垃圾，在计算机中是一个常用的术语，用来打印一些信息。——译者注

```
def dump(func):
    "打印输入参数和输出值"
    def wrapped(*args, **kwargs):
        print("Function name: %s" % func.__name__)
        print("Input arguments: %s" % ' '.join(map(str, args)))
        print("Input keyword arguments: %s" % kwargs.items())
        output = func(*args, **kwargs)
        print("Output:", output)
        return output
    return wrapped
```

下面是被装饰函数。这个函数名为 `double()`，接收带名称和不带名称的数值参数，把它们的值乘 2 并放到一个列表中返回：

```
from dump1 import dump

@dump
def double(*args, **kwargs):
    "每个参数乘2"
    output_list = [ 2 * arg for arg in args ]
    output_dict =  { k:2*v for k,v in kwargs.items() }
    return output_list, output_dict

if __name__ == '__main__':
    output = double(3, 5, first=100, next=98.6, last=-40)
```

运行结果：

```
$ python test_dump.py

Function name: double
Input arguments: 3 5
Input keyword arguments: dict_items([('last', -40), ('first', 100),
 ('next', 98.6)])
Output: ([6, 10], {'last': -80, 'first': 200, 'next': 197.2})
```

12.8 使用pdb进行调试

上面提到的技能很有用，但仍然无法代替真正的调试器。大多数 IDE 都自带了调试器，有各种各样的特性和用户界面。这里，我会介绍标准的 Python 调试器 `pdb`（https://docs.python.org/3/library/pdb.html）。

> 如果使用 `-i` 标志运行程序，出现错误时 Python 会自动进入交互式解释器。

下面来看一个在处理某些数据时存在 bug 的程序，这种 bug 很难发现。这是计算机早期出现的一个真实的 bug，困扰了程序员们很长时间。

我们需要从一个文件中读出国家和它们的首都，它们由逗号分割：首都，国家。它们的大

小写可能不正确，在输出时需要修复这个问题。哦，还有可能出现多余的空白，需要正确处理。最后，虽然程序知道是否到达文件结尾，出于某些原因，我们的领导希望在遇到单词 `quit` 时终止程序（大小写可能不正确）。下面是数据文件示例：

```
France, Paris
venuzuela,caracas
  LithuniA,vilnius
     quit
```

我们来设计一下算法（解决问题的方法）。下面是伪代码，它看起来像一个程序，其实只是用普通的语言来描述逻辑，还需要转换成真实的程序。程序员喜欢 Python 的一个原因就是它*看起来很像伪代码*，因此把伪代码转换成真正的程序比较简单：

```
for each line in the text file:
    read the line
    strip leading and trailing spaces
    if 'quit' occurs in the lower-case copy of the line:
        stop
    else:
        split the country and capital by the comma character
        trim any leading and trailing spaces
        convert the country and capital to titlecase
        print the capital, a comma, and the country
```

为了满足要求，需要去掉名字首尾的空格，还需要使用小写形式来和 `quit` 进行比较，并把城市和国家名转换成首字母大写。据此，我们可以快速写出 capitals.py，看起来应该可以正常工作：

```
def process_cities(filename):
    with open(filename, 'rt') as file:
        for line in file:
            line = line.strip()
            if 'quit' in line.lower():
                return
            country, city = line.split(',')
            city = city.strip()
            country = country.strip()
            print(city.title(), country.title(), sep=',')

if __name__ == '__main__':
    import sys
    process_cities(sys.argv[1])
```

使用之前准备好的示例数据文件来测试一下。预备、瞄准、发射：

```
$ python capitals.py  cities1.csv
Paris,France
Caracas,Venuzuela
Vilnius,Lithunia
```

很好！通过测试，我们把它放到生产环境中测试一下，使用下面的数据文件来处理首都和国家名称，直到出错：

```
argentina,buenos aires
bolivia,la paz
brazil,brasilia
chile,santiago
colombia,Bogotá
ecuador,quito
falkland islands,stanley
french guiana,cayenne
guyana,georgetown
paraguay,Asunción
peru,lima
suriname,paramaribo
uruguay,montevideo
venezuela,caracas
quit
```

运行程序，只打印出 5 行内容，但是数据文件中有 15 行：

```
$ python capitals.py  cities2.csv
Buenos Aires,Argentina
La Paz,Bolivia
Brazilia,Brazil
Santiago,Chile
Bogotá,Colombia
```

发生了什么？我们可以修改 capitals.py，在一些可能出错的地方添加 `print()` 语句，不过这里尝试一下调试器。

如果要使用调试器，需要在命令行中使用 **`-m pdb`** 来导入 `pdb` 模块，如下所示：

```
$ python -m pdb capitals.py cities2.csv

> /Users/williamlubanovic/book/capitals.py(1)<module>()
-> def process_cities(filename):
(Pdb)
```

这条命令会启动程序并停在第一行。如果你输入 `c`（继续），程序会一直运行下去，直到正常结束或者出现错误：

```
(Pdb) c

Buenos Aires,Argentina
La Paz,Bolivia
Brazilia,Brazil
Santiago,Chile
Bogotá,Colombia
The program finished and will be restarted
> /Users/williamlubanovic/book/capitals.py(1)<module>()
-> def process_cities(filename):
```

程序正常结束，和没使用调试器之前一样。我们再试一次，使用一些命令来缩小问题出现的范围。看起来这是一个逻辑错误，而不是语法错误或者异常（否则会打印出错误信息）。

输入 **`s`**（单步）来一行一行执行 Python 代码。这会单步执行所有 Python 代码：你自己的、标准库的、你用到的其他模块的。使用 `s` 时也会进入函数内部并继续单步执行。输入 **`n`**

（下一个）也可以单步执行，但是不会进入函数；如果遇到一个函数，n 会执行整个函数并前进到下一行代码。因此，不确定问题在哪时使用 s，确定问题不在函数中时使用 n，函数很长时尤其有用。通常来说，你需要单步执行自己的代码，跳过库代码，因为后者往往已经通过测试。我们在程序开头使用 s 来进入函数 process_cities()：

```
(Pdb) s

 > /Users/williamlubanovic/book/capitals.py(12)<module>()
 -> if __name__ == '__main__':

(Pdb) s

 > /Users/williamlubanovic/book/capitals.py(13)<module>()
 -> import sys

(Pdb) s

 > /Users/williamlubanovic/book/capitals.py(14)<module>()
 -> process_cities(sys.argv[1])

(Pdb) s

 --Call--
 > /Users/williamlubanovic/book/capitals.py(1)process_cities()
 -> def process_cities(filename):

(Pdb) s

 > /Users/williamlubanovic/book/capitals.py(2)process_cities()
 -> with open(filename, 'rt') as file:
```

输入 **l**（列表）来查看之后的几行：

```
 (Pdb) l

  1      def process_cities(filename):
  2  ->     with open(filename, 'rt') as file:
  3             for line in file:
  4                 line = line.strip()
  5                 if 'quit' in line.lower():
  6                     return
  7                 country, city = line.split(',')
  8                 city = city.strip()
  9                 country = country.strip()
 10                 print(city.title(), country.title(), sep=',')
 11
(Pdb)
```

箭头（->）指示当前行。

我们可以继续使用 s 或者 n，看看是否能发现问题，不过这里使用调试器的另一个重要特性：断点。断点会把程序暂停在你指定的位置。在本例中，我们想知道 process_cities() 为什么在读完所有输入之前退出。第 3 行（for line in file:）会读入输入文件的每一行，看起来没什么问题。函数中能在读入所有数据之前的地方只有第 6 行（return）。我们

在第 6 行设置一个断点：

```
(Pdb) b 6

Breakpoint 1 at /Users/williamlubanovic/book/capitals.py:6
```

接着继续运行程序，直到停在断点或者读入所有内容之后正常退出：

```
(Pdb) c

 Buenos Aires,Argentina
 La Paz,Bolivia
 Brasilia,Brazil
 Santiago,Chile
 Bogotá,Colombia
 > /Users/williamlubanovic/book/capitals.py(6)process_cities()
 -> return
```

它在第 6 行的断点停下了，也就是说，程序在处理完哥伦比亚之后就准备退出了。打印 `line` 的值，看看读入的是什么：

```
 (Pdb) p line

'ecuador,quito'
```

它有什么特别的吗？没有啊。

真的吗？ *quit*o ？我们的领导一定不希望正常数据中的字符串 `quit` 终止程序运行，看起来用它当哨兵值（表示终止运行）似乎是一个愚蠢的主意。你应该马上告诉他，我在这儿等你。

如果现在你还没丢掉工作，可以使用 `b` 命令查看所有的断点：

```
(Pdb) b

Num Type         Disp Enb   Where
1   breakpoint   keep yes   at /Users/williamlubanovic/book/capitals.py:6
    breakpoint already hit 1 time
```

`l` 命令会显示代码行，当前行（`->`）和断点（`B`）。`l` 命令默认会显示从上次使用 `l` 之后直到现在的所有代码，可以包含一个可选的起始行（这里从第 1 行开始）：

```
(Pdb) l 1

  1       def process_cities(filename):
  2          with open(filename, 'rt') as file:
  3              for line in file:
  4                  line = line.strip()
  5                  if 'quit' in line.lower():
  6 B->                  return
  7                  country, city = line.split(',')
  8                  city = city.strip()
  9                  country = country.strip()
 10                  print(city.title(), country.title(), sep=',')
 11
```

好了，修复一下 `quit` 问题，只让它在匹配整行时退出：

```
def process_cities(filename):
    with open(filename, 'rt') as file:
        for line in file:
            line = line.strip()
            if 'quit' == line.lower():
                return
            country, city = line.split(',')
            city = city.strip()
            country = country.strip()
            print(city.title(), country.title(), sep=',')

if __name__ == '__main__':
    import sys
    process_cities(sys.argv[1])
```

再试一次：

```
$ python capitals2.py cities2.csv

Buenos Aires,Argentina
La Paz,Bolivia
Brasilia,Brazil
Santiago,Chile
Bogotá,Colombia
Quito,Ecuador
Stanley,Falkland Islands
Cayenne,French Guiana
Georgetown,Guyana
Asunción,Paraguay
Lima,Peru
Paramaribo,Suriname
Montevideo,Uruguay
Caracas,Venezuela
```

本章简单介绍了一下调试器，只是告诉你调试器可以做什么以及最常用的命令。

记住：测试越多，调试越少。

12.9　记录错误日志

有时候，你需要使用比 `print()` 更高端的工具来记录日志。日志通常是系统中的一个文件，用于持续记录信息。信息中通常会包含很多有用的内容，比如时间戳或者运行程序的用户的名字。通常来说，日志每天会被旋转（重命名）并压缩，这样它们就不会占用太多磁盘空间。如果程序出错，你可以查看对应的日志文件来了解发送了什么。异常信息非常重要，因为它们会告诉你出错的行数和原因。

Python 标准库模块中有一个 `logging`（https://docs.python.org/3/library/logging.html）。我发现关于它的许多描述都很难懂。使用一段时间之后可以更好地理解，但是对于新手来说，有点太复杂了。`logging` 模块包含以下的内容：

- 你想保存到日志中的消息；

- 不同的优先级以及对应的函数：`debug()`、`info()`、`warn()`、`error()` 和 `critical()`；
- 一个或多个 logger 对象，主要通过它们使用模块；
- 把消息写入终端、文件、数据库或者其他地方的 handler；
- 创建输出的 formatter；
- 基于输入进行筛选的过滤器。

下面是最简单的日志示例，导入模块并使用它的函数：

```
>>> import logging
>>> logging.debug("Looks like rain")
>>> logging.info("And hail")
>>> logging.warn("Did I hear thunder?")
WARNING:root:Did I hear thunder?
>>> logging.error("Was that lightning?")
ERROR:root:Was that lightning?
>>> logging.critical("Stop fencing and get inside!")
CRITICAL:root:Stop fencing and get inside!
```

看到了吗，`debug()` 和 `info()` 什么都没做，另外两个函数在每条消息之前打印出来级别 `:root:`。到目前为止，它们很像有个性的 `print()` 语句，有些还对我们抱有敌意[2]。

不过它们很有用。你可以在一个日志文件中搜索指定级别的消息，通过比较时间戳来看，在服务器崩溃之前发生了什么。

查看文档之后我们找到了第一个问题的答案（第二个稍后介绍）：默认的优先级是 `WARNING`，所以前两个函数没有输出。当调用第一个函数（`logging.debug()`）时它就被锁定了。我们可以使用 `basicConfig()` 来设置默认的级别。`DEBUG` 是最低一级，因此下面的例子会打印出所有级别的消息：

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG)
>>> logging.debug("It's raining again")
DEBUG:root:It's raining again
>>> logging.info("With hail the size of hailstones")
INFO:root:With hail the size of hailstones
```

上面的例子都是直接使用默认的 `logging` 函数，没有创建 logger 对象。每个 logger 都有一个名称。创建一个名为 `bunyan` 的 logger：

```
>>> import logging
>>> logging.basicConfig(level='DEBUG')
>>> logger = logging.getLogger('bunyan')
>>> logger.debug('Timber!')
DEBUG:bunyan:Timber!
```

如果 logger 名称中包含点号，会生成不同层级的 logger，每层可以有不同的属性。也就是说，名为 `quark` 的 logger 比名为 `quark.charmed` 的层级更高。特殊的 root logger 在最顶层，它的名字是 `''`。

注 2：因为有些消息的语气很严厉。——译者注

到目前为止，我们只是打印出消息，和 print() 的差别并不大。可以使用 handler 把消息输出到不同的地方。最常见的是写入日志文件，如下所示：

```
>>> import logging
>>> logging.basicConfig(level='DEBUG', filename='blue_ox.log')
>>> logger = logging.getLogger('bunyan')
>>> logger.debug("Where's my axe?")
>>> logger.warn("I need my axe")
>>>
```

啊哈，这些内容并没有出现在屏幕上，而是写入了文件 blue_ox.log：

```
DEBUG:bunyan:Where's my axe?
WARNING:bunyan:I need my axe
```

调用 basicConfig() 时使用 filename 参数会创建一个 FileHandler 并对 logger 进行设置。logging 模块至少包含 15 种 handler，比如电子邮件、Web 服务器、屏幕和文件。

最后，你可以控制消息的格式。在我们的第一个例子中，默认的格式是这样：

```
WARNING:root:消息...
```

如果给 basicConfig() 传入 format 字符串，可以改变格式：

```
>>> import logging
>>> fmt = '%(asctime)s %(levelname)s %(lineno)s %(message)s'
>>> logging.basicConfig(level='DEBUG', format=fmt)
>>> logger = logging.getLogger('bunyan')
>>> logger.error("Where's my other plaid shirt?")
2014-04-08 23:13:59,899 ERROR 1 Where's my other plaid shirt?
```

我们修改了消息的格式并让 logger 把消息输出到屏幕。logging 模块可以识别出 fmt 格式化字符串中的变量名。我们使用了 asctime（一个包含日期和时间的 ISO 8601 字符串）、levelname、lineno（行号）和 message。它们都是内置的变量，你也可以创建自定义变量。

logging 还有许多这里没有提到的功能，比如说，你可以同时把消息输出到多个位置，每个位置都有不同的优先级和格式。这个包的扩展性很强，不过有时候会牺牲一些简洁性。

12.10 优化代码

一般来说，Python 已经足够快了，直到它不够快的那一刻之前。大多数情况下，你都可以使用更好的算法或者数据结构来加速，关键是知道把这些技巧用在哪里。即使是经验丰富的程序员也常常会犯错误。你需要像裁缝一样耐心，在裁剪之前认真测量。下面来介绍一下计时器。

12.10.1 测量时间

之前已经看到过，time 模块中的 time 函数会返回一个浮点数，表示当前的纪元时间。测量时间最简单的方法就是先获取当前时间、做一些事情、获取新的时间，然后用新时间减去初始时间。具体代码是 time1.py：

```
from time import time

t1 = time()
num = 5
num *= 2
print(time() - t1)
```

在本例中，我们测量了把 5 赋值给变量 num 并给它乘以 2 所需要的时间。这并不是真正的基准测试，只是告诉你如何测量 Python 代码的执行时间。把它运行几次，看看变化幅度：

```
$ python time1.py
2.1457672119140625e-06
$ python time1.py
2.1457672119140625e-06
$ python time1.py
2.1457672119140625e-06
$ python time1.py
1.9073486328125e-06
$ python time1.py
3.0994415283203125e-06
```

大概是两百万或者三百万分之一秒。试试更慢的代码，比如 sleep。如果睡眠一秒，计时器应该比一秒稍大一些。把下面的代码保存为 time2.py：

```
from time import time, sleep

t1 = time()
sleep(1.0)
print(time() - t1)
```

为了检验我们的猜测，多次运行这段代码：

```
$ python time2.py
1.000797986984253
$ python time2.py
1.0010130405426025
$ python time2.py
1.0010390281677246
```

意料之中，它需要运行一秒多一点。如果不是这个结果，那计时器和 sleep() 肯定有一个出了问题。

有一种更简单的方法来测量代码片段的执行时间：标准模块 timeit（https://docs.python.org/3/library/timeit.html）。它有一个函数叫作（你应该能猜到）timeit()，这个函数会运行你的测试代码 count 次并打印结果。调用形式：timeit.timeit(code，number，count)。

在本节的例子中，code 需要放在引号中，这样它们会在 timeit() 内部运行，而不是直接运行。（下一节会看到如何用 timeit() 来测量函数的运行时间。）运行一次之前的示例代码并计时。把下面的代码存为 timeit1.py：

```
from timeit import timeit
print(timeit('num = 5; num *= 2', number=1))
```

运行几次：

```
$ python timeit1.py
2.5600020308047533e-06
$ python timeit1.py
1.9020008039660752e-06
$ python timeit1.py
1.7380007193423808e-06
```

和上面差不多，这两行代码需要大约两百万分之一秒来运行。我们可以使用 `timeit` 模块的 `repeat()` 函数的 `repeat` 参数来运行多次。把下面的代码保存为 timeit2.py：

```
from timeit import repeat
print(repeat('num = 5; num *= 2', number=1, repeat=3))
```

运行一下：

```
$ python timeit2.py
[1.691998477326706e-06, 4.070025170221925e-07, 2.4700057110749185e-07]
```

第一次运行用了大约两百万分之一秒，第二和第三次运行快了很多。为什么？原因可能有很多，比如说，由于我们测试的代码片段太小，它的运行速度取决于计算机同时在做的其他事情、Python 系统对计算的优化方式以及其他许多事。

也有可能只是碰巧。下面来尝试一些相比变量赋值和 `sleep` 更加真实的代码。我们会测量一些代码并比较不同算法（程序逻辑）和数据结构（存储方式）的效率。

12.10.2　算法和数据结构

Python 之禅（http://legacy.python.org/dev/peps/pep-0020/）提到，*应该有一种，最好只有一种，明显的解决方法*。不幸的是，有时候并没有那么明显，你需要比较各种方案。举例来说，如果要构建一个列表，使用 `for` 循环好还是列表解析好？我们如何定义更好？是更快、更容易理解、占用内存更少还是更具 Python 风格？

在接下来的练习中，我们会用不同的方式构建列表，比较速度、可读性和 Python 风格。下面是 time_lists.py：

```
from timeit import timeit

def make_list_1():
    result = []
    for value in range(1000):
        result.append(value)
    return result

def make_list_2():
    result = [value for value in range(1000)]
    return result

print('make_list_1 takes', timeit(make_list_1, number=1000), 'seconds')
print('make_list_2 takes', timeit(make_list_2, number=1000), 'seconds')
```

在每个函数中，我们都向列表添加了 1000 个元素，还分别调用两个函数 1000 次。注意，这里调用 `timeit()` 时，传入的第一个参数是函数名不是字符串形式的代码。运行一下：

```
$ python time_lists.py
make_list_1 takes 0.14117428699682932 seconds
make_list_2 takes 0.06174145900149597 seconds
```

列表解析至少比用 `append()` 添加元素快两倍。通常来说，列表解析要比手动添加快。

可以使用这个方法来加速你的代码。

12.10.3　Cython、NumPy和C扩展

如果你已经尽了最大努力仍然无法达到想要的速度，还有一些方法可以选择。

Cython（http://cython.org/）混合了 Python 和 C，它的设计目的是把带有性能注释的 Python 代码翻译成 C 代码。这些注释非常简单，比如声明一些变量、函数参数或者函数返回值的类型。对于科学上的数字运算循环来说，添加这些注释之后会让程序快很多，速度能达到之前的一千倍。可以在 Cython wiki（https://github.com/cython/cython/wiki）查看文档和示例。

NumPy 是 Python 的一个数学库，它是用 C 语言编写的，运行速度很快。你可以在附录 C 中阅读 NumPy 的相关信息。

为了提高性能并且方便使用，Python 的很多代码和标准库都是用 C 写成并用 Python 进行封装。这些钩子可以直接在你的程序中使用。如果你熟悉 C 和 Python 并且真的想提高程序性能，可以写一个 C 扩展，这样做难度很大但是效果很好。

12.10.4　PyPy

大约 20 年前，Java 刚出现时，速度就像一只得了关节炎的雪纳瑞一样慢。当 Sun 和其他公司看到它的价值之后，它们投入了几百万美元来优化 Java 的解释器和底层的 Java 虚拟机（JVM），借鉴了其他语言（比如 Smalltalk 和 LISP）的许多技术。微软也投入了很大精力来优化它的 C# 语言和 .NET 虚拟机。

Python 不属于任何人，因此没人投入这么多努力来提高它的速度。你使用的很可能是标准的 Python 实现。它是由 C 写成，通常被称为 CPython（和 Cython 不一样）。

和 PHP、Perl 甚至 Java 一样，Python 并不会被编译成机器语言，而是被翻译成中间语言（通常被称为*字节码*或者 *p- 代码*），然后被*虚拟机*解释执行。

PyPy（http://pypy.org/）是一个新出现的 Python 解释器，实现了许多 Java 中的加速技术。它的基准测试（http://speed.pypy.org/）显示 PyPy 几乎完全超越了 CPython——平均快 6 倍，最高快 20 倍。它支持 Python 2 和 Python 3，你可以下载并用它代替 CPython。PyPy 在不断改进，某一天可能会取代 CPython。可以阅读官网的最新发布内容，看看是否可以用在你的项目上。

12.11　源码控制

当你在一个小团队或者小项目中工作时，通常很容易跟踪自己的改动，直到你犯了很愚蠢的错误并花费很多时间来修复它。源码控制系统会保护你的代码不被破坏（比如被你

自己的愚蠢错误）。如果你和其他开发者在一个团队工作，源码控制就变得极其重要。这个领域有很多收费和开源的包可以用。在 Python 所处的开源领域，最流行的是 Mercurial 和 Git，它们都是分布式版本控制系统，会生成代码仓库的多个副本。早期的系统，比如 Subversion，是工作在单个服务器上的。

12.11.1 Mercurial

Mercurial（http://mercurial.selenic.com/）是用 Python 写成的。它很容易学习，有很多有用的子命令，比如从 Mercurial 仓库下载代码、添加文件、提交改动、从不同的源合并改动。bitbucket（https://bitbucket.org/）和其他网站（https://mercurial.selenic.com/wiki/MercurialHosting）提供了免费或者收费的托管服务。

12.11.2 Git

Git（http://git-scm.com/）最初是用于 Linux 内核开发，但是现在基本上已经统治了开源领域。它和 Mercurial 很像，不过有些人发现它更难精通。GitHub（https://github.com/）是最大的 Git 托管平台，拥有超过一百万个仓库。除了 GitHub，还有一些其他的 Git 托管平台（https://git.wiki.kernel.org/index.php/GitHosting）。

本书中的程序示例都放在 GitHub（https://github.com/madscheme/introducing-python）的一个公共 Git 仓库中。如果你的计算机上有 `git` 程序，可以使用下面的命令来下载这些程序：

```
$ git clone https://github.com/madscheme/introducing-python
```

也可以点击 GitHub 页面上的按钮来下载代码：

- 点击“Clone in Desktop”来打开桌面版的 `git`，如果之前已经安装过；
- 点击“Download ZIP”来下载压缩归档的程序。

如果你没有 `git`，但是想尝试一下，可以阅读安装教程（http://git-scm.com/book/en/v2/Getting-Started-Installing-Git）。我会介绍命令行版本的 `git`，不过你也可以试试 GitHub 这样的网站，它们会提供额外的服务并且有时候会更好用。`git` 有许多特性，但有时候不太直观。

下面来尝试一下 `git`。我们不会太深入，不过会展示一些命令和它们的输出。

创建一个新目录并进入：

```
$ mkdir newdir
$ cd newdir
```

在当前目录 newdir 中创建一个本地 Git 仓库：

```
$ git init

Initialized empty Git repository in /Users/williamlubanovic/newdir/.git/
```

在 newdir 中创建一个 Python 文件 test.py，内容如下所示：

```
print('Oops')
```

把这个文件添加到 Git 仓库：

```
$ git add test.py
```

Git 先生，感觉如何？

```
$ git status

On branch master

Initial commit

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

    new file:   test.py
```

这表示 test.py 已经是本地仓库的一部分，但是它的改动还没有被提交。我们来提交一下：

```
$ git commit -m "simple print program"

 [master (root-commit) 52d60d7] my first commit
  1 file changed, 1 insertion(+)
  create mode 100644 test.py
```

`-m "my first commit"` 是你的提交说明。如果忽略，说明 `git` 会打开一个编辑器并要求你输入说明。它会被记录到这个文件在 `git` 中的提交历史里。

看看现在的状态：

```
$ git status

 On branch master
 nothing to commit, working directory clean
```

好，所有改动都已经提交。这表示我们可以进行任何改动并且不用担心丢失原始版本。修改一下 test.py，把 `Oops` 改成 `Ops!` 并保存文件：

```
print('Ops!')
```

再来看看 `git` 的感觉：

```
$ git status

On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

    modified:   test.py

no changes added to commit (use "git add" and/or "git commit -a")
```

使用 `git diff` 来查看上次提交之后的改动：

```
$ git diff

 diff --git a/test.py b/test.py
 index 76b8c39..62782b2 100644
 --- a/test.py
 +++ b/test.py
 @@ -1 +1 @@
 -print('Oops')
 +print('Ops!')
```

如果尝试提交改动，git 会抱怨：

```
$ git commit -m "change the print string"

On branch master
Changes not staged for commit:
    modified:   test.py

no changes added to commit
```

staged for commit 的意思是需要先 add 文件，可以把这个动作翻译成：嘿，Git，往这儿看：

```
$ git add test.py
```

也可以输入 git add . 来添加当前目录下的所有改动文件。如果你修改了很多文件并且想要提交所有改动，这个命令会非常方便。现在提交改动：

```
$ git commit -m "my first change"

 [master e1e11ec] my first change
  1 file changed, 1 insertion(+), 1 deletion(-)
```

如果想查看你对 test.py 做的所有事情，按照时间倒序排列，可以使用 git log：

```
$ git log test.py

 commit e1e11ecf802ae1a78debe6193c552dcd15ca160a
 Author: William Lubanovic <bill@madscheme.com>
 Date:   Tue May 13 23:34:59 2014 -0500

     change the print string

 commit 52d60d76594a62299f6fd561b2446c8b1227cfe1
 Author: William Lubanovic <bill@madscheme.com>
 Date:   Tue May 13 23:26:14 2014 -0500

     simple print program
```

12.12 复制本书代码

你可以下载本书中所有程序的代码。查看 Git 仓库（https://github.com/madscheme/introducing-python）并按照教程把仓库复制到你的本地电脑上。如果你有 git，可以运行命令 git clone https://github.com/madscheme/introducing-python 来复制仓库。你也可以下载 zip 格式的文件。

12.13 更多内容

这本书只是入门教程，大部分内容是基础知识，实践方面内容不多。下面，我会推荐一些有用的 Python 教程。

12.13.1 书

下面是我觉得非常有用的图书，从入门到精通都有，涵盖了 Python 2 和 Python 3：

- Barry, Paul. *Head First Python*. O'Reilly, 2010.
- Beazley, David M. *Python Essential Reference* (4th Edition). Addison-Wesley, 2009.
- Beazley, David M. and Brian K. Jones. *Python Cookbook* (3rd Edition). O'Reilly, 2013.
- Chun, Wesley. *Core Python Applications Programming* (3rd Edition). Prentice Hall, 2012.
- McKinney, Wes. *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*. O'Reilly, 2012.
- Summerfield, Mark. *Python in Practice: Create Better Programs Using Concurrency, Libraries, and Patterns.* Addison-Wesley, 2013.

当然，除此之外还有很多（https://wiki.python.org/moin/PythonBooks）。

12.13.2 网站

可以在下面这些网站上找到很多有用的教程：

- Zed Shaw 的 Learn Python the Hard Way（http://learnpythonthehardway.org/book/）
- Mark Pilgrim 的 Dive Into Python 3（http://www.diveintopython3.net/）
- Michael Driscoll 的 Mouse Vs. Python（http://www.blog.pythonlibrary.org/）

如果你对 Python 世界的新东西很感兴趣，可以看看下面这些网站：

- comp.lang.python（https://groups.google.com/forum/#!forum/comp.lang.python）
- comp.lang.python.announce（https://groups.google.com/forum/#!forum/comp.lang.python.announce）
- python subreddit（http://www.reddit.com/r/python）
- Planet Python（http://planet.python.org/）

最后是一些下载 Python 代码的好地方：

- The Python Package Index（https://pypi.python.org/pypi）
- stackoverflow Python questions（http://stackoverflow.com/questions/tagged/python）
- ActiveState Python recipes（http://code.activestate.com/recipes/langs/python/）
- Python packages trending on GitHub（https://github.com/trending?l=python）

12.13.3 社区

有很多计算机社区：热心的、好辩的、沉闷的、时髦的、无聊的，等等。Python 社区非常

友好和民主。你可以基于位置发现 Python 社区——meetups（http://python.meetup.com/）和遍布全世界（https://wiki.python.org/moin/LocalUserGroups）的本地用户社区。还有一些分布式的基于共同兴趣的社区。举例来说，PyLadies（http://www.pyladies.com/）是由那些对 Python 和开源感兴趣的女性组成。

12.13.4　大会

Python 有遍布全世界（https://www.python.org/community/workshops/）的大会（http://www.pycon.org/）和实践课程，每年在南美（https://us.pycon.org/2015/）和欧洲（https://europython.eu/en/）举办的大会规模最大。

12.14　后续内容

等等，还没结束！附录 A、附录 B 和附录 C 分别介绍了 Python 在艺术、商业和科学方面的应用。你至少能找到一个感兴趣的包。

互联网上有很多闪闪发亮的东西，只有你自己知道，对你来说，哪些是装饰品哪些是银弹[3]。即使你现在还没有遇到狼人，也应该在口袋中备一些银弹，以防万一。

书的最后是每章结尾练习的答案、安装 Python 和其相关内容的详细教程，以及我觉得很有用的速查表。虽然你对这些东西已经很熟悉了，不过万一需要可以去书后查阅。

注 3：传说中，银弹是狼人的克星。在《人月神话》中，作者用银弹比喻能快速解决难题的方法。这里作者的意思也是如此，学会 Python 以备不时之需。——译者注

附录A

Python的艺术

> “艺术就是艺术，对吧？同样，水就是水！东就是东，西就是西。如果你把越橘炖得像苹果酱一样，那么它尝起来就比大黄更像西梅。”
>
> ——Groucho Marx

你可能是个艺术家、抑或是音乐家，或者你只是想尝试一些不同的能激发无限创造力的事情。

前三个附录收录了人们在不同场合、不同领域借助 Python 进行的探索。如果你对其中的某些领域感兴趣，可能会从中获取些许灵感，或产生动手尝试的冲动。

A.1　2D图形

几乎所有的计算机编程语言或多或少都会被应用到图形处理中。尽管本章用到的许多功能强大的平台，出于效率考虑，都是由 C 或 C++ 编写的，但它们大都喜欢用 Python 编写的库来扩展其功能。我们先来看一些 2D 成像库。

A.1.1　标准库

标准库中与图形相关的库并不多，这里介绍其中的两个。

- `imghdr`

  这个模块用于检测图片文件的文件格式。

- `colorsys`

  这个模块用于在不同的颜色系统（RGB、YIQ、HSV 以及 HLS）间进行转换。

如果你已经把 O’Reilly 的标志下载到本地并存储为 oreilly.png，可以直接运行下面这段代码：

```
>>> import imghdr
>>> imghdr.what('oreilly.png')
'png'
```

为了在 Python 中对图片进行更高级的处理，我们需要安装一些第三方包。先来看看有哪些包可用。

A.1.2 PIL和Pillow

多年以来，尽管 Python 图片库（Python Image Library，PIL，http://effbot.org/imagingbook/pil-index.htm）并不属于 Python 的标准库，但它一直是 Python 中最著名的 2D 图片处理库。由于它在包安装器 pip 之前就已出现，因此我们无法用便捷的 pip 命令进行安装。对此，有人创建了一个友好的项目分支 Pillow（http://pillow.readthedocs.org/）。Pillow 向后兼容 PIL，并且文档更为规范，因此这里我们选择使用 Pillow 作为替代。

安装过程非常简单，在终端中执行下面的命令即可：

```
$ pip install Pillow
```

如果你曾经安装过 libjpeg、libfreetype 和 zlib 这几个系统包，它们会被 Pillow 检测出来并且直接使用，不再重复安装。你可以在安装页（http://pillow.readthedocs.org/en/latest/installation.html）了解更多安装细节。

打开一个图片文件：

```
>>> from PIL import Image
>>> img = Image.open('oreilly.png')
>>> img.format
'PNG'
>>> img.size
(154, 141)
>>> img.mode
'RGB'
```

尽管包的名字是 Pillow，但为了让它与旧的 PIL 兼容，我们需要将它引用为 PIL。

使用 Image 对象的 show() 方法可以在屏幕上显示图片，但为此你需要先安装 ImageMagick 包。下一小节会介绍这个包，到时候试着运行下面的代码：

```
>>> img.show()
```

执行上述代码后，屏幕上会跳出一个新窗口，如图 A-1 所示。（截图是在 Mac OS X 系统下截取的，show() 实际上打开了 Mac 中预览程序来显示图片。不同系统下的图片浏览窗口可能有所不同。）

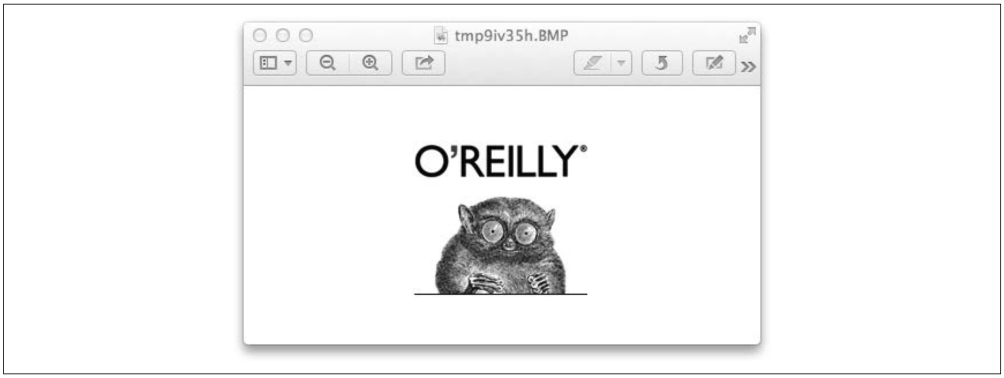


图 A-1：使用 Python 库打开的图片

将放入内存中的图片裁剪一下，把结果存储为一个新的对象 `img2` 并重新显示它。图片总是以水平坐标值（x）和垂直坐标值（y）计量，它们以图片的一角作为参考。这个角被称作*原点*，它的 x 坐标和 y 坐标都是 0。我们使用的库中，原点（0, 0）定义为图片的左上角，x 向右增加，y 向下增加。想要给 `crop()` 方法传入 4 个参数：左侧距 x（55）、顶距 y（70）、右侧距 x（85）以及底距 y（100），需要先将它们按顺序封装在元组中。

```
>>> crop = (55, 70, 85, 100)
>>> img2 = img.crop(crop)
>>> img2.show()
```

结果如图 A-2 所示。

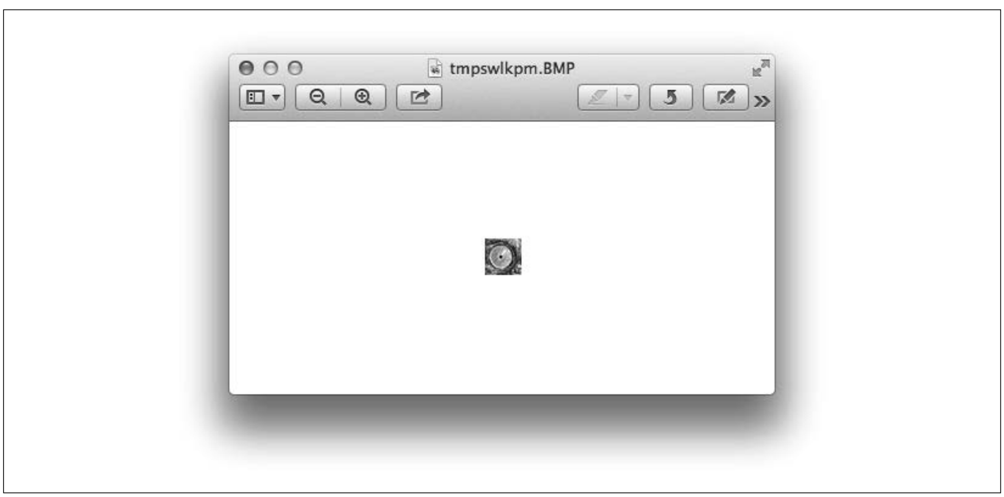


图 A-2：裁剪后的图片

使用 `save` 方法存储图片。它接收一个文件名参数以及一个可选参数——图片类型。如果文件名中有扩展名，库函数会直接使用这个扩展名作为图片的格式，当然你也可以显式指定使用的格式。执行下面的代码可以将裁剪后的图片存储为 GIF 文件：

```
>>> img2.save('cropped.gif', 'GIF')
>>> img3 = Image.open('cropped.gif')
>>> img3.format
'GIF'
>>> img3.size
(30, 30)
```

我们来美化一下我们的小吉祥物吧。首先将胡须图片（http://www.pinkmoustache.net/wp-content/uploads/2009/12/moustaches.png）下载到本地，存储为 moustaches.png。接着将它加载到内存中，进行适当的裁剪，然后把它置于之前图片的上层：

```
>>> mustache = Image.open('moustaches.png')
>>> handlebar = mustache.crop( (316, 282, 394, 310) )
>>> handlebar.size
(78, 28)
>>> img.paste(handlebar, (45, 90) )
>>> img.show()
```

图 A-3 展示了最佳效果。



图 A-3：全新帅气的吉祥物

如果胡须图片的背景是透明的话效果会更好。不如把这个当作练习吧！如果你想要把上面的例子做到最好，查看一下 Pillow 教程（http://pillow.readthedocs.org/en/latest/handbook/tutorial.html）中关于透明度和 alpha 值的相关内容，然后自己动手试试吧！

A.1.3　ImageMagick

ImageMagick（http://www.imagemagick.org/script/index.php）是一套用于转换、修改、显示 2D 位图的程序。它已经有 20 余年的历史了。许多第三方 Python 库都会应用到 ImageMagick C 库，其中最近的一个支持 Python 3 的库是 wand（http://docs.wand-py.org/en/0.4.0/）。执行下面的语句来安装它：

```
$ pip install Wand
```

你可以使用 wand 做许多 Pillow 能做到的事情：

```
>>> from wand.image import Image
>>> from wand.display import display
>>>
>>> img = Image(filename='oreilly.png')
>>> img.size
(154, 141)
>>> img.format
'PNG'
```

和 Pillow 类似，下面的语句用于显示图片：

```
>>> display(img)
```

wand 包含旋转、调整大小、添加文本、画线、格式转化等强大的功能，这些功能你也可以在 Pillow 中找到。Wand 和 Pillow 的 API 和文档都非常规范好用。

A.2　图形用户界面

GUI 这个名词虽然包括图形这个词，但它的重点更在于用户界面（展示数据用的小控件、输入的方法、菜单、按钮以及包含所有这些的窗口等）上。

GUI 编程（https://wiki.python.org/moin/GuiProgramming）的 wiki 页以及 FAQ（https://docs.python.org/3.3/faq/gui.html）列出了许可用 Python 编写的 GUI。我们从唯一一个内置于标准库的 Tkinter（https://wiki.python.org/moin/TkInter）开始介绍。它功能不多，但可以在所有平台上创建符合接近原生界面的窗口以及控件。

下面是一个极简的 Tkinter 程序，它可以在新建的窗口中显示我们最爱的小吉祥物，它有着曲棍球似的眼睛：

```
>>> import tkinter
>>> from PIL import Image, ImageTk
>>>
>>> main = tkinter.Tk()
>>> img = Image.open('oreilly.png')
>>> tkimg = ImageTk.PhotoImage(img)
>>> tkinter.Label(main, image=tkimg).pack()
>>> main.mainloop()
```

注意，上面的代码用到了 PIL/Pillow 中的一些模块。你应该又一次见到了 O’Reilly 的标志，如图 A-4 所示。

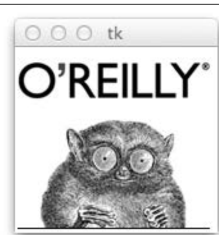


图 A-4：使用 Tkinter 库显示的图片

想要关闭窗口的话只需点击关闭按钮或者关闭 Python 解释器即可。

你可以在 tkinter wiki（http://tkinter.unpythonic.net/wiki/）和 Python wiki（https://wiki.python.org/moin/TkInter）上看到更多关于 Tkinter 的信息。现在来看一些不包含在标准库中的 GUI。

- Qt（http://qt-project.org/）

  Qt 是一款由挪威 Trolltech 公司开发的专业级 GUI 及应用工具包，已有 20 多年的历史。像著名的 Google Earth、Maya 以及 Skype 都是由 Qt 协助完成的。它还被用于一款桌面版 Linux——KDE 的开发。Qt 有两个主要的 Python 库版本：其中 PySide（http://wiki.qt.io/PySide）是免费的（LGPL 许可），PyQt（http://www.riverbankcomputing.com/software/pyqt/intro）则是基于 GPL 许可付费使用的。你可以在这里看到它们之间的差别（http://wiki.qt.io/Differences_Between_PySide_and_PyQt），可以从 PyPi（https://pypi.python.org/pypi/PySide）或 Qt（http://qt-project.org/wiki/Get-PySide）网站下载 Qt 并阅读相关教程（http://qt-project.org/wiki/PySide_Tutorials），可以在线（http://www.riverbankcomputing.com/software/pyqt/download5）免费下载 Qt 主体。

- GTK+（http://www.gtk.org/）

  GTK+ 是 Qt 的一个强劲对手，它也被广泛应用于包括 GIMP 和 Gnome（一款桌面版 Linux）在内的许多应用（http://gtk-apps.org/）的开发。它与 Python 通过 PyGTK（http://www.pygtk.org/）进行绑定。你可以访问 PyGTK 的官网（http://www.pygtk.org/downloads.html）下载代码、阅读相关文档（http://python-gtk-3-tutorial.readthedocs.org/en/latest/）。

- WxPython（http://www.wxpython.org/）

  用于将 Python 与 WxWidgets（http://www.wxwidgets.org/）进行绑定。WxWidgets 也是一款功能强大的 UI 工具包，你可以从网上（http://wxpython.org/download.php）免费下载。

- Kivy（http://kivy.org/）

  Kivy 是一个用于搭建跨平台——PC 端（Windows、OS X、Linux）及移动端（Android、iOS）——多媒体 UI 的现代库。它包含对于多点触控的支持。你可以从 Kivy 官网（http://kivy.org/#download）下载所有平台对应的库文件。它还提供了不同平台的应用开发教程（http://kivy.org/docs/gettingstarted/intro.html）。

- 网络

  像 Qt 这样的框架依赖的都是本地控件，然而也有一些其他的框架使用的是网络。毕竟网络是一种通用的 GUI，它包含图形（SVG）、文本（HTML）甚至现在还可以包含多媒体（HTML5）。使用 Python 编写的基于网络的 GUI 工具包括 RCTK（Remote Control Toolkit，https://code.google.com/p/rctk/）以及 Muntjac（http://muntiacus.org/）。你可以创建支持任意前端（基于浏览器）和后端（网络服务器）组合的网络程序。在瘦客户端模式下，后端会完成大部分工作。而如果前端起支配作用的话，我们称之为厚客户端、胖客户端或富客户端（最后一个词显然有拍马屁的嫌疑）。端与端之间的通信常使用 RESTful API、AJAX 以及 JSON。

A.3 3D图形和动画

现在几乎每一部电影的片尾字幕里都包含了许多制作特效和动画的人名。许多大型工作室——华特迪士尼动画、ILM、维塔数码、梦工厂、皮克斯——都会雇不少具有 Python 经验的人。你可以上网搜索“python 动画职位”看看，或者到 vfxjobs 上搜索关键词“python”看看有多少职位空缺。

如果你喜欢探索 Python，喜欢 3D、动画、多媒体以及游戏的话，应该试试 Panda3D（http://www.panda3d.org/）。它开源、完全免费，你甚至可以用它搭建商业应用。你可以从 Panda3D 官网（http://www.panda3d.org/download.php?sdk）上下载与你电脑系统对应的版本。如果你想要测试一下它自带的示例程序，需要先将终端的路径切换到 /Developer/Examples/Panda3D。这个目录中的每一个子目录都包含了一个或多个 .py 文件。使用 Panda3D 的 ppython 命令来运行这些示例程序。例如：

```
$ cd /Developer/Examples/Panda3D
$ cd Ball-in-Maze/
$ ppython Tut-Ball-in-Maze.py
DirectStart: Starting the game.
Known pipe types:
  osxGraphicsPipe
(all display modules loaded.)
```

一个类似图 A-5 的窗口会打开。



图 A-5：使用 Panda3D 库显示的图片

你可以使用鼠标来倾斜盒子让球在迷宫中前进。

如果一切运行正常，Panda3D 的环境搭建也一切顺利，那么恭喜你，现在就可以开始使用这个库了。

下面是一个来自于 Panda3D 文档的示例程序，将它保存为 panda1.py：

```
from direct.showbase.ShowBase import ShowBase

class MyApp(ShowBase):

    def __init__(self):
        ShowBase.__init__(self)

        # 加载环境模型
        self.environ = self.loader.loadModel("models/environment")
        # 为模型重定父级以渲染
        self.environ.reparentTo(self.render)
        # 对模型应用尺度和位置变换
        self.environ.setScale(0.25, 0.25, 0.25)
        self.environ.setPos(-8, 42, 0)

app = MyApp()
app.run()
```

使用下面的命令运行它：

```
$ ppython panda1.py
Known pipe types:
  osxGraphicsPipe
(all display modules loaded.)
```

运行后会弹出一个窗口，如图 A-6 所示。当前的石头和树是悬浮在半空中的。点击 `Next` 按钮，按照教程继续，它会引导你解决这个问题。



图 A-6：使用 Panda3D 库显示的适应窗口大小的图片

下面是其他一些可以在 Python 中使用的 3D 包。

- Blender（http://www.blender.org/）

  Blender 是一个免费的 3D 动画和游戏的生成器。当你从 http://www.blender.org/download/ 下载并安装了它后，它会绑定一个自带的 Python 3。

- Maya（http://www.autodesk.com/products/maya/overview）

  Maya 是一个商用的 3-D 动画和图形处理系统。它也会绑定一个自带的 Python，目前绑定的是 Python 2.6。Chad Vernon 编写了 *Python Scripting for Maya Artists*（http://www.chadvernon.com/blog/resources/python-scripting-for-maya-artists/），你可以在网上免费下载。如果你在网上搜索 Python 和 Maya，可以找到许多其他资源，包括一些视频资料，它们有些是免费的，有些可能是付费的。

- Houdini（https://www.sidefx.com/）

  Houdini 是一个商用系统，但你可以下载它的一个免费版本 Apprentice。和其他动画包一样，它也有自己绑定的 Python（http://www.sidefx.com/docs/houdini13.0/hom/）。

A.4　平面图、曲线图和可视化

Python 是目前绘制平面图、曲线图以及进行数据可视化的最佳解决方法。它在科学研究中尤其受到欢迎，你可以在附录 C 了解相关应用。Python 的官方网站上关于可视化有一些概论性内容（https://wiki.python.org/moin/NumericAndScientific/Plotting）。在这里我想花一点时间做一些额外介绍。

A.4.1　matplotlib

使用下面的命令可以免费安装 `matplotlib` 2D 绘图库：

```
$ pip install matplotlib
```

官网图片库（http://matplotlib.org/gallery.html）中的例子展示了 `matplotlib` 的应用范围之广。我们使用和之前一样的程序来展示图片（如图 A-7 所示），这里只是为了看看需要编写的代码以及显示结果如何：

```
import matplotlib.pyplot as plot
import matplotlib.image as image

img = image.imread('oreilly.png')
plot.imshow(img)
plot.show()
```

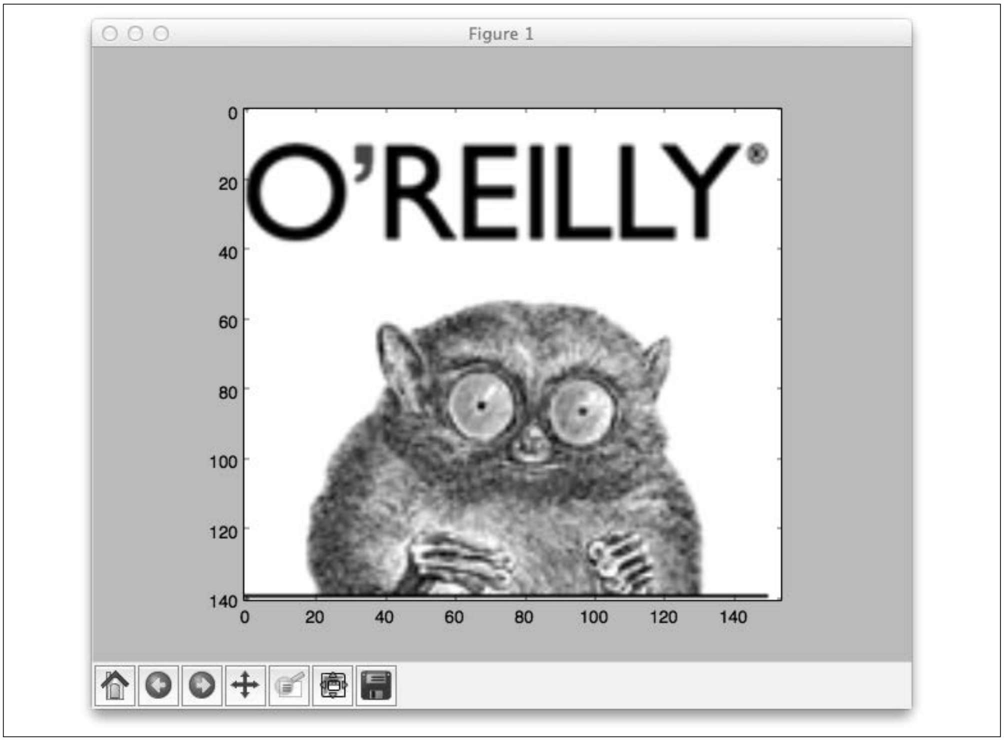


图 A-7：使用 `matplotlib` 库显示的图片

你可以在附录 C 中看到更多关于 `matplotlib` 的内容，它与 NumPy 以及其他一些研究性应用关系十分密切。

A.4.2 bokeh

在网络发展初期，为了在线显示图形，开发者们需要在服务器端将其生成好并给浏览器返回对应的 URL。但如今，JavaScript 的表现能力越来越强，一些像 D3 之类的在客户端生成图形的工具也越发流行。前几页中，我曾提到过可以使用 Python 搭建前 – 后端框架用于展示图形和 GUI。一款名为 `bokeh`（http://bokeh.pydata.org/en/latest/）的新工具将 Python 的能力（大数据集、易于使用）与 JavaScript（交互性、图形绘制延迟小）结合了起来。它致力于大数据集的快速可视化。

如果你已经安装过 `bokeh` 所依赖的库（NumPy、Pandas 和 Redis），那么现在可以通过下面一行简单的命令来安装 `bokeh`：

```
$ pip install bokeh
```

（你可以在附录 C 看到 NumPy 和 Pandas 的作用。）

或者，你也可以从 Bokeh 官网（http://bokeh.pydata.org/en/latest/docs/quickstart.html）下载并一口气安装所有所需内容。尽管 `matplotlib` 是运行在服务器上的，`bokeh` 主要在浏览器

上运行，它可以充分利用最近客户端能力的提升。点击官网图片库（http://bokeh.pydata.org/en/latest/docs/gallery.html）中的图片可以进行交互式查看，并查阅实现该效果所用的 Python 代码。

A.5 游戏

Python 非常适合数据处理，在这一附录中，你又发现了 Python 在处理多媒体上的强大功能。那么关于游戏呢？

没错，碰巧 Python 还是一个绝佳的游戏开发平台。使用 Python 开发游戏的书籍不计其数，例如下面两本：

- Al Sweigart 编写的 *Invent Your Own Computer Games with Python*（http://inventwithpython.com）
- Horst Jens 编写的 *The Python Game Book*（一本 wiki 文档式的书，http://thepythongamebook.com/en:start）

Python 的 wiki 页（https://wiki.python.org/moin/PythonGames）上包含了更多关于使用 Python 开发游戏的讨论。

最著名的 Python 游戏开发平台要数 pygame（http://pygame.org/news.html）了。你可以从 Pygame 官网（http://pygame.org/download.shtml）上下载对应平台的可执行安装包进行安装，并阅读 pummel the chimp 游戏的逐行实例（http://pygame.org/docs/tut/chimp/ChimpLineByLine.html）。

A.6 音频和音乐

Python 还能做什么？音频、音乐？或者让我的猫学会唱 Jingle Bell ?

好吧，其实只有前两个能做到。

标准库的多媒体服务（http://docs.python.org/3/library/mm.html）下有一些基本的音频模块，当然还有其他许多第三方模块（https://wiki.python.org/moin/Audio）可以使用。

下面这些库可以帮你生成音乐：

- pyknon（https://github.com/kroger/pyknon）用在 Pedro Kroger（CreateSpace）的 *Music for Geeks and Neds*（https://github.com/kroger/pyknon）一书中；
- mingus（https://code.google.com/p/mingus/）是一个音序器，用于读取 / 生成 MIDI 文件；
- remix（http://echonest.github.io/remix/python.html），就像名字本身暗示的一样，是一组混音用的 API。它的应用之一是 morecowbell.dj（http://morecowbell.dj）这个网站，它能往你上传的歌曲中添加牛铃声；
- sebastian（https://github.com/jtauber/sebastian/）是一个用于音乐理论分析的库；
- Piano（http://jugad2.blogspot.com/2013/04/play-piano-on-your-computer-with-python.html）可以让你用键盘来弹奏不同调性（例如 C、D、E、F、A、B）的音乐。

最后，下面列出的库可以帮助你整理你的音乐集或访问音乐数据：

- Beets（http://beets.radbox.org/）用于管理音乐集；
- Echonest（http://developer.echonest.com/）是对音乐元数据进行访问用的 API；
- Monstermash（http://karlgrz.com/monstermash-flask-zeromq-and-echonest-remix/）用于对音乐片段进行拼接合成，它建立在 Echonest、Flask、ZeroMQ 以及 Amazon EC2 上；
- Shiva（https://hacks.mozilla.org/2013/03/shiva-more-than-a-restful-api-to-your-music-collection/）是一个用来访问音乐集的 RESTful API 和服务器（https://github.com/tooxie/shiva-server）；
- 下载一个 album art（http://jameh.github.io/mpd-album-art/）来查找匹配你的音乐。

附录B

工作中的Python

“生意！”鬼魂攥紧手凄凉地说道，“我为什么不把人看成是我的生意的一部分……”

——查尔斯·狄更斯的《圣诞颂歌》

男性商务人士总是习惯一身西装戴着领带。但不知为何，每当他们要真正开始干活时总是磨磨蹭蹭的，他们会先把夹克扔到椅背上，然后解开领带，卷起袖口，再冲些咖啡。与此同时，女员工往往已经默默地把活都干完了。也许会有一杯拿铁相伴。

在实际商业活动中，我们会使用到前面曾介绍过的所有技术——数据库、互联网、系统和网络。Python 凭借着它极高的生产力被广泛应用于企业（http://opensource.com/life/13/11/python-enterprise-jessica-mckellar）和创业公司（http://opensource.com/business/13/12/why-python-perfect-startups）。

在商业界，人们曾在很长的一段时间内寻求能够一击解决遗产问题（legacy problem）的银色子弹。这些遗产问题包括：互不兼容的文件格式、晦涩难懂的网络协议、开发语言独占以及文档准确性普遍不高。然而今天，我们已经可以看到一些科学技术开始可以协同工作、成规模发展。通过采用下面这些建议，公司可以创建更快捷、更便宜且更易于扩展的应用：

- 像 Python 一样的动态语言
- 将互联网作为通用 GUI
- 使用 RESTful API 作为独立于语言的服务接口
- 关系型数据库和 NoSQL 数据库
- 大数据及大数据分析
- 通过“云”进行部署，节省开支

B.1 Microsoft Office套件

当今商业的正常运行对 Microsoft Office 应用及其文件格式有着很强的依赖性。Python 中有一些可以对 Office 文件进行处理的库，但它们大都没什么名气，甚至有些连像样的文档都没有。下面是一些可以用来处理 Office 文件的 Python 库的例子。

- docx（https://pypi.python.org/pypi/docx）

  这个库可以用于创建、读取、写入 Microsoft Office Word 2007 及以上使用的 .docx 文件。

- python-excel（http://www.python-excel.org/）

  这个库有一份 PDF 教程（http://www.simplistix.co.uk/presentations/python-excel.pdf），介绍了如何使用 `xlrd`、`xlwt` 和 `xlutils` 模块。Excel 还可以对逗号分隔值（Comma-Saparated Value，CSV）格式的文件进行读写。你应该已经知道如何使用 `csv` 模块处理 CSV 文件了。

- oletools（http://www.decalage.info/python/oletools）

  这个库可以从 Office 格式的文件中抽取数据。

下面这些模块可自动化处理 Windows 应用：

- `pywin32`（http://sourceforge.net/projects/pywin32/）

  这个模块可以自动化处理许多 Windows 应用[1]。但是它仅限在 Python 2 环境中使用，而且文档不全。你可以参考博客：http://www.galalaly.me/index.php/2011/09/use-python-to-parse-microsoft-word-documents-using-pywin32-library/ 和 http://www.blog.pythonlibrary.org/2010/07/16/python-and-microsoft-office-using-pywin32/。

- `pywinauto`（https://code.google.com/p/pywinauto/）

  这个模块也是用于自动化处理 Windows 应用的，同样限制于 Python 2。使用方法可以参考这篇博客：https://techsaju.wordpress.com/2013/03/06/using-python-pywinauto-for-automating-windows-gui-applications/。

- `swapy`（https://code.google.com/p/swapy/）

  `swapy` 可以直接根据用户的原始操作生成对应的 `pywinauto` 的代码。

OpenOffice（http://www.openoffice.org/）是一款开源的 Office 应用的替代品。它可以在 Linux、Unix、Windows 以及 OS X 上运行，可以对 Office 格式的文件进行读写操作。OpenOffice 自身也绑定了供自己使用的 Python 3。你可以通过 PyUNO 库（http://www.openoffice.org/udk/python/python-bridge.html）对 OpenOffice 进行 Python 编程（https://wiki.openoffice.org/wiki/Python）。

OpenOffice 是由 Sun Microsystems 开发的，当甲骨文将它收购之后，许多用户害怕未来会无法免费使用这款产品。LibreOffice（https://www.libreoffice.org/）因此横空出世。DocumentHacker（https://documenthacker.files.wordpress.com/2013/07/writing_

注 1：类似于 Windows 中宏的作用，它可以记录你执行的可重复性操作便于以后复用。——译者注

documents-_for_software_engineers_v0002.pdf）描述了如何使用 Python 的 UNO 库对 LibreOffice 进行编程。

为了对 Office 文件进行读写操作，OpenOffice 和 LibreOffice 不得不对 Microsoft 的文件格式进行逆向解析，这个过程难度可不小。通用 Office 转换器（Universal Office Converter，http://dag.wiee.rs/home-made/unoconv/）就依赖于 OpenOffice 和 LibreOffice 中的 UNO 库。这个转换器可以转换多种文件格式：文档、电子表格、图表和演示文稿。

你可以使用 python-magic（https://github.com/ahupp/python-magic）对格式未知文件中的某些字节序进行分析从而猜测出它的具体格式。

python open document 库（http://appyframework.org/pod.html）允许你根据模板格式提供对应的 Python 代码来创建动态文档。

Adobe 的 PDF 格式在商业中也十分常见，尽管它并不属于 Micorsoft 格式家族。ReportLab（http://www.reportlab.com/opensource/）是一个基于 Python 的 PDF 文件生成器，它包含一个开源版本和一个商用版本。如果你需要用 Python 编辑一个 PDF 文件的话，也许可以从 StackOverflow（http://stackoverflow.com/questions/1180115/add-text-to-existing-pdf-using-python）获取些帮助。

B.2 执行商业任务

无论需要做什么事情，你几乎都可以找到对应的 Python 模块来帮助你。访问 PyPI（https://pypi.python.org/pypi）并将你想要搜索的内容敲进搜索框就可以了。你可能会对其中一些与商业任务相关的模块有兴趣：

- 使用 Fedex（https://github.com/gtaylor/python-fedex）或 UPS（https://github.com/openlabs/PyUPS）运送货物。
- 使用 stamps.com（https://github.com/jzempel/stamps）API 邮递。
- 阅读关于 Python for business intelligence（http://www.slideshare.net/Stiivi/python-business-intelligence-pydata-2012-talk）的讨论。
- 爱乐压咖啡机在 Anoka 的畅销是正常的消费行为还是有什么内情？ Cubes（http://cubes.databrewery.org/）是一个可以进行联机分析处理（Online Analytical Processing，OLAP）的网络服务器，它同时还是一个数据浏览器，可以帮你对上述问题进行分析。
- OpenERP（https://www.openerp.com/）是一个大型的商用企业资源计划（Enterprise Resource Planning，ERP）系统。它是由 Python 和 JavaScript 编写的，拥有数千个扩展模块。

B.3 处理商业数据

商业界对数据情有独钟。但不幸的是，它们总是莫名其妙地制造麻烦让数据变得难以使用。

电子表格是一项伟大的发明，随着时间的推移商业界对它愈发痴迷。许多非程序员被诱骗去编程，仅仅因为他们编码的被称作宏（macro）而不是程序。但就如同宇宙在不断扩张一样，数据也在飞速膨胀。旧版本的 Excel 最多支持 65 536 行数据，即使是新版本也只不

过能支持百万行。当一个企业所拥有的数据量超过了单一计算机所能承载的最大值，就像企业员工数过百一样——瞬间你会发现旧框架变得不再适用，你亟需新的层级结构、新的媒介以及新的通信方式。

海量数据处理程序的诞生并不是由单机上庞大的数据量导致的，而是由于不同来源的数据倾注到单一业务中聚合导致的。关系型数据库可以同时处理百万行数据而不崩溃，但这种性能也只有在集中写入或更新数据时才能达到。旧的纯文本文件或二进制文件可以增长到 GB 级别的大小，但如果你需要一口气对它进行全面处理的话，就必须有足够大的内存（否则只能分块读取到内存中进行处理）。传统的桌面程序并不是为解决这些问题而设计的。但像 Google 和亚马逊这样的公司则不得不去寻找大规模处理数据的解决方案。Netflix（http://techblog.netflix.com/2013/03/python-at-netflix.html）就是一个例子，它建立在亚马逊的 AWS 云系统上，使用 Python 将 RESTful API、信息安全、部署调度、数据库有机地结合了起来，提供了一套完整的解决方案。

B.3.1 提取、转换、加载

数据就像一座冰山，复杂的操作都埋藏在水下。其中占最大部分的、同时也是最重要的是关于获取数据的操作。企业中对于处理数据常用的术语是提取、转换、加载（Extracting，Transforming，Loading），也就是所谓的 ETL。类似的同义词还有 data munging 或 data wrangling，给人一种驯服凶猛野兽的感觉[2]，比喻得还算恰当。现在看来，这似乎只是一个解决了的工程问题，但其实如何优雅巧妙地处理海量数据还是需要一定艺术灵感的。附录 C 会涉及更多关于数据科学的内容，这是许多开发者需要花费大部分时间挣扎的地方。

如果你看过《绿野仙踪》，可能还记得（除了飞天猴外）快结尾的部分：善良的巫师告诉多萝西，其实她只要敲敲她的红鞋子就可以回到堪萨斯的家了。尽管当时我还小，但还是有一种“拖了这么久，她终于告诉多萝西了！”的感觉。事后回顾，我意识到，如果女巫早点就把这个诀窍告诉多萝西的话，那这电影就没法拍了。

但在这里我们讨论的不是电影，而是商业。在商业，能把任务尽快完成是一件好事。所以，我想要在这里告诉你们一些窍门。你每天工作中用来处理数据的大多数工具都在这本书的前面有所涉及，包括高度抽象的数据结构，例如字典、对象，成千上万个标准库和第三方库以及一个满是高手的社区，离你只有 Google 一下的距离。

如果你是一个从事商务的程序员，那你每天的工作流程几乎都逃不开下面这些步骤：

(1) 从奇怪格式的文件和数据库中抽取出目标数据；
(2) 清洗数据，包括各种苦差事，你需要和各种各样的对象打交道；
(3) 进行转换工作，例如日期的转换、时间的转换以及字符集的转换；
(4) 真正开始处理数据；
(5) 将结果数据存储在文件或者数据库中；
(6) 回到第一步重新继续；给数据抹上肥皂，清洗，一遍又一遍。

举个例子：设想你需要将电子表格中的一些数据挪到数据库里。一种可行的方式是将电子

注 2：munging 和 wrangling 有争吵、斗争的意思。——译者注

表格存储为 CSV 格式，然后用第 8 章介绍过的 Python 库进行处理。或者也可以搜索看看有没有可以直接读取二进制电子表格文件的 Python 模块。你的手会自然而然地在 Google 里敲入 `python excel`，找到像 Working with Excel files in Python（http://www.python-excel.org/）这样的网站。接着你可以使用 `pip` 安装你感兴趣的包，然后定位 Python 的数据库驱动为后续工作做好准备。第 8 章也提到过 SQLAlchemy 以及直接与底层数据库打交道的驱动。准备工作就绪，现在我们可以在这两者之间编写点代码了，这里才是 Python 的数据结构以及库真正开始帮你节省时间的地方。

首先来看一个例子，之后我们会尝试使用一些库来简化其中某些步骤。我们需要读取一个 CSV 文件，然后以某一列的值作为键，将具有相同键的数据进行聚合（这里使用加法聚合）。如果在 SQL 中进行这些操作的话，会用到 `SELECT`、`JOIN`、`GROUP BY` 等命令。

首先关于待处理文件，zoo.csv，包含以下几列内容：动物种类、咬伤游客的次数、处理咬伤需要缝合的针数以及为了防止游客告知当地媒体所支付的赔偿金。

```
animal,bites,stitches,hush
bear,1,35,300
marmoset,1,2,250
bear,2,42,500
elk,1,30,100
weasel,4,7,50
duck,2,0,10
```

我们想要了解到底哪种动物给我们带来的损失最多，因此需要计算在每一种动物身上花费的总的封口费数目（至于如何处理咬伤和如何缝针就交给实习医生了，不是我们程序员能力所及）。我们需要使用 8.2.1 节介绍过的 `csv` 模块以及 5.5.2 节介绍过的 `Counter`。将下面的代码存储为 zoo_counts.py：

```
import csv
from collections import Counter

counts = Counter()
with open('zoo.csv', 'rt') as fin:
    cin = csv.reader(fin)
    for num, row in enumerate(cin):
        if num > 0:
            counts[row[0]] += int(row[-1])
for animal, hush in counts.items():
    print("%10s %10s" % (animal, hush))
```

我们直接跳过第一行，因为第一行仅仅提供了每列的名称，没有任何有实际价值的数据。`counts` 是一个 `Counter` 对象，它首先会将每一种动物对应的值（封口费）初始化为 0。此外还对输出结果进行了右对齐调整。看一下输出结果：

```
$ python zoo_counts.py
      duck         10
       elk        100
      bear        800
    weasel         50
  marmoset        250
```

最让我们操心的果然是熊！它一直就是我们主要怀疑的对象，这下有数据为证了。

现在试着使用一个数据处理工具 Bubbles（http://bubbles.databrewery.org/）来重复上面的工作。你可以通过下面这条命令来安装它：

```
$ pip install bubbles
```

它依赖 SQLAlchemy，如果你没有安装过的话可以通过 `pip install sqlalchemy` 安装。下面是我们根据文档（http://pythonhosted.org/bubbles/install.html#quick-start）改写的测试程序（命名为 bubbles1.py）：

```
import bubbles

p = bubbles.Pipeline()
p.source(bubbles.data_object('csv_source', 'zoo.csv', infer_fields=True))
p.aggregate('animal', 'hush')
p.pretty_print()
```

真相只有一个！接下来就是见证真相的时刻了：

```
$ python bubbles1.py

 2014-03-11 19:46:36,806 DEBUG calling aggregate(rows)
 2014-03-11 19:46:36,807 INFO called aggregate(rows)
 2014-03-11 19:46:36,807 DEBUG calling pretty_print(records)
 +--------+--------+------------+
 |animal  |hush_sum|record_count|
 +--------+--------+------------+
 |duck    |      10|           1|
 |weasel  |      50|           1|
 |bear    |     800|           2|
 |elk     |     100|           1|
 |marmoset|     250|           1|
 +--------+--------+------------+
 2014-03-11 19:46:36,807 INFO called pretty_print(records)
```

Bubbles 的文档中提供了将调试信息隐藏掉的方法，甚至还包括改变输出表格格式的方法。

对比这两个例子，我们发现 `bubbles` 例子中我们仅仅使用了一个函数调用（`aggregate`）就完成了之前手动读取、计数 CSV 文件的全部工作。根据需求的不同，数据处理工具有时能为你节省出大量时间。

更现实的例子中，zoo 文件可能包含几千行记录（这个动物园还真是危险），甚至包含拼写错误（例如把 `bear` 写成了 `bare`），或者数字中无意添加了分隔符，等等。如果你想要更多关于如何使用 Python 和 Java 进行实际数据处理的例子，我推荐 Greg Wilson 的 *Data Crunching: Solve Everyday Problems Using Java, Python, and More*（Pragmatic Bookshelf，http://shop.oreilly.com/product/9780974514079.do）。

数据清理工具可以节省你许多时间，而 Python 中恰好就有许多这样的工具。再举个例子，PETL（http://petl.readthedocs.org/en/latest/）可以对表格中的数据进行行列提取以及重命名。它的相关产品（http://petl.readthedocs.org/en/latest/related_work.html）网页列出了许多有用的模块和产品。附录 C 里有一些关于常用的数据处理工具的讨论，例如 Pandas、

NumPy 以及 IPython。尽管这些工具现在仍主要被科学研究者所熟知，但它们在金融工作者以及数据处理员之间也开始愈加流行。2012 年 Pydata 会议上，AppData（http://www.computerworld.com/article/2492915/big-data/python--big-data-s-secret-power-tool.html）介绍了上述三款工具以及其他的一些 Python 工具是如何帮助我们每日处理 15TB 的数据的。毫无疑问，Python 有足够的能力处理海量的实际数据。

B.3.2 额外信息源

有时你可能需要使用一些来源于企业之外的数据。下面是一些可以使用的商业和政府提供的数据源。

- data.gov（https://www.data.gov/）
  数千个数据集和数据处理工具的门户。它的 API（https://www.data.gov/developers/apis）是搭建在一套 Python 数据管理系统——CKAN（http://ckan.org/）上的。
- Opening government with Python（http://sunlightfoundation.com/）
  从这里获取相关视频（https://www.youtube.com/watch?v=FTwjUL6Gq4A）及幻灯片（http://goo.gl/8Yh3s）。
- python-sunlight（http://python-sunlight.readthedocs.org/en/latest/）
  用于访问 Sunlight API（http://sunlightfoundation.com/api/）的库。
- froide（http://stefanw.github.io/froide/）
  一个基于 Django 的管理信息自由请求（Freedom of Information Request）的工具。
- 30 个寻找公开数据的网址（http://blog.visual.ly/data-sources/）
  一些有用的链接。

B.4 金融中的Python

最近，金融界对于 Python 的兴趣愈发浓烈。数据分析专家们（quant）正在搭建新一代的金融工具，他们或在附录 C 中提到的 Python 工具的基础上进行改写，或另起炉灶从头编写。

- Quantitative economics（http://quant-econ.net/）
  这是一个用于建立经济模型的工具，包含许多数学公式和 Python 代码。
- Python for finance（http://www.python-for-finance.com）
  以 Yves Hilpisch 的书 *Derivatives Analytics with Python: Data Analytics, Models, Simulation, Calibration, and Hedging*（Wiley）为代表，介绍 Python 在金融中的应用。
- Quantopian（https://www.quantopian.com/）
  Quantopian 是一个交互式网站，你可以在上面编写自己的 Python 代码来整理股票的历史数据，进行后验测试。
- PyAlgoTrade（http://gbeced.github.io/pyalgotrade/）
  这也是一个对股票进行后验测试的工具，只不过 PyAlgoTrade 运行在你自己的计算机

上，无需联网。

- Quandl（http://www.quandl.com/）
  你可以使用 Quandl 在数百万金融数据集中进行搜索。
- Ultra-finance（https://code.google.com/p/ultra-finance/）
  实时的股票信息库。
- *Python for Finance*（O'Reilly，http://shop.oreilly.com/product/0636920032441.do）
  Yves Hilpisch 的一本关于如何使用 Python 进行金融建模的书。

B.5 商业数据安全性

在商业中，安全性永远是一个重要的问题。关于这方面的内容可以写整整一本书，事实上也确实有许多关于此的书，因此在这里我们不多涉及，仅仅提几个与 Python 有关的小贴士。

11.2.6 节中介绍了 `scapy`，一款用 Python 编写的网络包内容查看器。它经常用于网络攻击分析中。

Python Security（http://www.pythonsecurity.org/）网站上有许多关于安全性的话题，包括 Python 中一些与安全加密相关的模块以及使用技巧。

TJ O'Connor 的书 *Violent Python: A Cookbook for Hackers, Forensic Analysts, Penetration Testers and Security Engineers*（http://shop.oreilly.com/product/9781597499576.do，Syngress）中广泛介绍了 Python 及计算机安全方面的内容。

B.6 地图

地图具有很高的商业价值，而 Python 又非常善于绘制地图，所以我们会在这方面多花点篇幅介绍。高层们都喜欢图表，如果你能迅速为你们企业的官网绘制出一张漂亮的展示地图，他们一定会对你有所好感。

在互联网发展初期，我就曾访问过 Xerox 旗下的一个地图制作网站，当时还处于试验阶段。而随后像 Google Maps 这样的大网站的上线带来了一场新的革命（同经典的那句"为什么我就没有想到这个点子挣上几百万呢？"）。再到现在，处处可见地图和基于地理位置的服务（location-based service），它们在移动设备中尤为有用。

关于地图的许多术语的意思都有所重叠，例如地图绘制（mapping）、制图学（cartography）、GIS（地理信息系统，Geographic Information System）、GPS（全球定位系统，Global Positioning System）、地理空间分析（geospatial analysis），等等。Geospatial Python（http://geospatialpython.com/2013_11_01_archive.html）上的博客中有一张描述"无所不能（800-pound gorilla）"的系统——GDAL/OGR、GEOS 以及 PROJ.4

（projections）——以及它周边系统的图片（图中用猩猩来表示这个系统[3]）。它们大多提供了 Python 接口。我们会对其中的一些进行讲解，就从最简单的地图格式讲起。

B.6.1 格式

地图的世界中有许多不同的表示格式：向量（线）、栅格（图片）、元数据（词）以及它们之间的不同组合。

Esri 是开发地理系统的先驱，在 20 年前发明了 shapefile 格式。一个 shapefile 格式的文件通常包含了多个子文件，其中至少需要包含下面这些：

- .shp
  “形状”（向量）信息
- .shx
  形状索引
- .dbf
  属性数据库

下面列出了一些 Python 中用于处理 shapefile 文件的有用的模块：

- `pyshp`（https://code.google.com/p/pyshp/）是一个纯 Python 编写的 shapefile 库；
- `shapely`（http://toblerity.org/shapely/）可以处理与地理位置相关的问题，例如：“这座小镇里哪些房子位于 50 年洪水线上？”
- `fiona`（https://github.com/Toblerity/Fiona）包含了用于处理 shapefile 以及其他向量格式文件的 OGR 库；
- `kartograph`（http://kartograph.org/）可以将 shapefile 格式的文件转化为便于在服务器端或客户端上使用的 SVG 格式；
- `basemap`（http://matplotlib.org/basemap/）使用 `matplotlib` 在地图上绘制 2-D 数据；
- `cartopy`（http://scitools.org.uk/cartopy/docs/latest/）使用 `matplotlib` 和 `shapely` 绘制地图。

为了便于之后讲解示例，我们需要先获取一个 shapefile 格式的文件。访问 Natural Earth 上的 1:110m Cultural Vectors page（http://www.naturalearthdata.com/downloads/110m-cultural-vectors/），在“Admin 1 - States and Provinces”一栏下，点击绿色的 download states and provinces（http://www.naturalearthdata.com/http/www.naturalearthdata.com/download/110m/cultural/ne_110m_admin_1_states_provinces.zip）框可以下载得到一个压缩文件。下载完成后将它解压，你应该可以看到下面列出的这些文件：

```
ne_110m_admin_1_states_provinces_shp.README.html
ne_110m_admin_1_states_provinces_shp.sbn
ne_110m_admin_1_states_provinces_shp.VERSION.txt
ne_110m_admin_1_states_provinces_shp.sbx
ne_110m_admin_1_states_provinces_shp.dbf
ne_110m_admin_1_states_provinces_shp.shp
```

注 3：前文 800-pound gorilla 在英文中有无所不能、强大的意思，图片使了双关。——译者注

```
ne_110m_admin_1_states_provinces_shp.prj
ne_110m_admin_1_states_provinces_shp.shx
```

后面的例子会使用到这些文件。

B.6.2　绘制地图

你需要安装下面这个库来读取 shapefile 文件：

```
$ pip install pyshp
```

现在该看看程序了。下面是我根据 Geospatial Python 博客（http://geospatialpython.com/2010/12/rasterizing-shapefiles.html）修改得到的代码，map1.py：

```
def display_shapefile(name, iwidth=500, iheight=500):
    import shapefile
    from PIL import Image, ImageDraw
    r = shapefile.Reader(name)
    mleft, mbottom, mright, mtop = r.bbox
    # map units
    mwidth = mright - mleft
    mheight = mtop - mbottom
    # scale map units to image units
    hscale = iwidth/mwidth
    vscale = iheight/mheight
    img = Image.new("RGB", (iwidth, iheight), "white")
    draw = ImageDraw.Draw(img)
    for shape in r.shapes():
        pixels = [
            (int(iwidth - ((mright - x) * hscale)), int((mtop - y) * vscale))
            for x, y in shape.points]
        if shape.shapeType == shapefile.POLYGON:
            draw.polygon(pixels, outline='black')
        elif shape.shapeType == shapefile.POLYLINE:
            draw.line(pixels, fill='black')
    img.show()

if __name__ == '__main__':
    import sys
    display_shapefile(sys.argv[1], 700, 700)
```

上面的代码读取了 shapefile 文件，并遍历了其中的图形库。我们只关注其中两种图形类型：首尾相连的多边形和首尾不相连的折线。上面的代码仅仅是我简单看了一眼 pyshp 的文档一遍编写出来的，所以我并不百分百保证它能正常运行。但往往万事开头难，不要怕出错，出错了慢慢修改即可。

好了，运行一下上面的代码。需要传入的参数是 shapefile 文件的原始文件名，不添加任何扩展名：

```
$ python map1.py ne_110m_admin_1_states_provinces_shp
```

你应该会看到类似下面图 B-1 所示的结果。

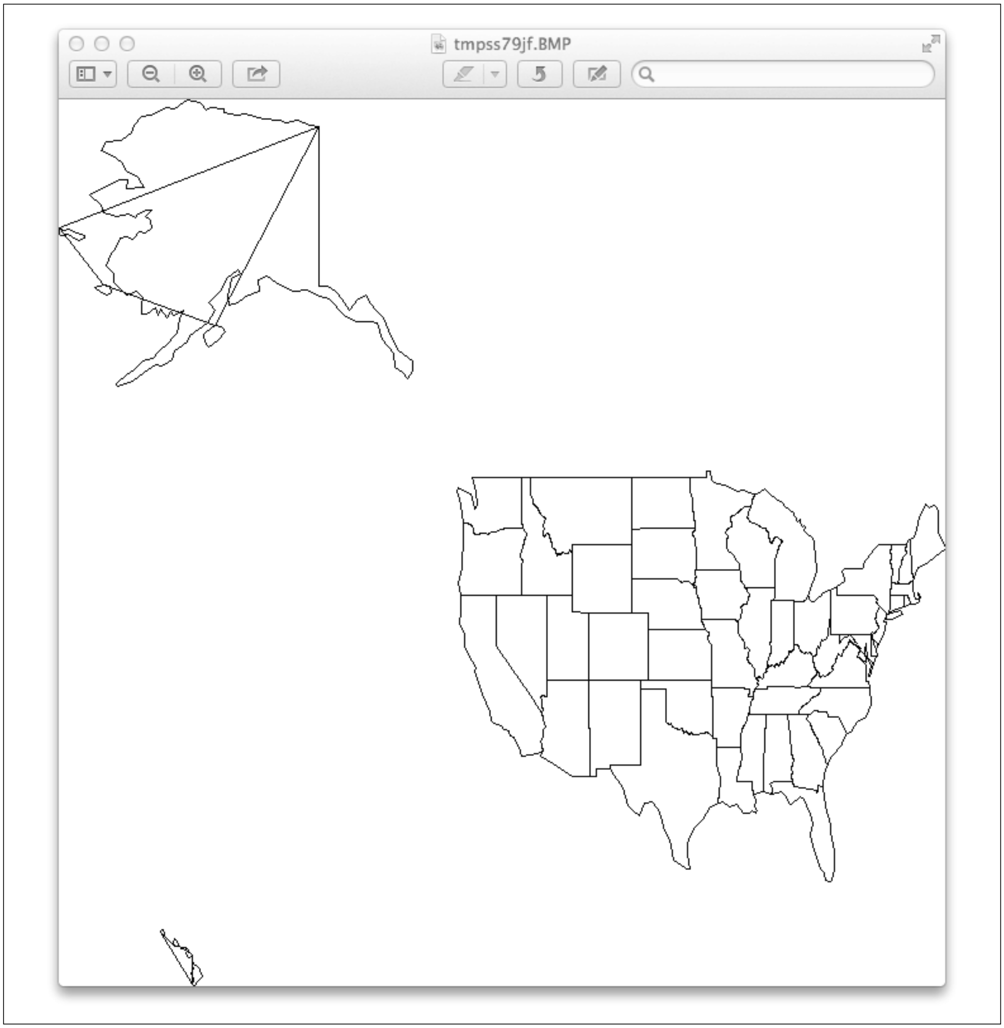


图 B-1：原始地图

恩，它确实成功画出了一幅类似美国地图的图，但似乎存在些问题：

- 阿拉斯加和夏威夷的地图像是被顽皮的猫拽了一条线缠绕了起来，这显然是个 bug；
- 地图被压扁了，我们需要对原始图做映射（projection）处理；
- 图片不太美观，我们需要做些样式（style）修改。

关于上述第一点：显然是因为我草草编写的代码逻辑出现了问题，应该怎么改呢？第 12 章曾介绍了一些开发建议，包括如何调试 bug，但这里我们可以考虑一些其他途径。比如可以不断添加一些测试代码直到定位并解决问题，或者可以干脆使用别的绘图库试试。也许有些抽象能力更高的库可以一口气解决上面所有问题（零散杂乱的线、压缩的外观以及原始丑陋的样式）。

下面是其他一些可选的 Python 绘图软件。

- basemap（http://matplotlib.org/basemap/）
  基于 `matplotlib`，可以一层一层地绘图以及添加数据。
- mapnik（http://mapnik.org/）
  以 C++ 编写的库，提供了 Python 接口，便于快速开发。可以绘制矢量地图（线）和栅格地图（图片）。
- tilemill（https://www.mapbox.com/tilemill/）
  一个基于 `mapnik` 的地图设计平台。
- Vincent（http://vincent.readthedocs.org/en/latest/）
  一个从 Python 到 Vega（一款 JavaScript 可视化工具）的翻译器。你可以阅读 *Mapping data in Python with pandas and vicent*（http://wrobstory.github.io/2013/10/mapping-data-python.html）这篇博客学习如何使用。
- Python for ArcGIS（http://resources.arcgis.com/en/communities/python/）
  链接到 Esri 的商业产品 ArcGIS 中包含的 Python 资源。
- 使用 Python 进行空间分析（https://complex.luxbulb.org/howto/spatial-analysis-python）
  链接到相关的教程、包以及视频。
- 使用 Python 处理地理空间数据（https://conference.scipy.org/scipy2013/tutorial_detail.php?id=110）
  链接到相关视频展示。
- 使用 Python 绘制地图（http://sensitivecities.com/so-youd-like-to-make-a-map-using-python-EN.html）
  使用 `pandas`、`matplotlib`、`shapely` 以及其他 Python 模块绘制历史性地标的地图。
- *Python Geospatial Development*（http://shop.oreilly.com/product/9781782161523.do）
  Eric Westra 的一本书，介绍了如何使用 `mapnik` 以及其他工具。
- *Learning Geospatial Analysis with Python*（http://shop.oreilly.com/product/9781783281138.do）
  Joel Lawhead 的一本书，回顾了与地理空间相关的算法、格式以及库。

上面提到的模块都能绘制出精美的地图，但它们的安装、学习难度都比较大。有些模块甚至依赖于你没有见过的库，例如 `numpy` 和 `pandas`。使用它们带来的收益能否大于付出？作为开发者，我们总是不得不在掌握的信息不完整的情况下被迫做出取舍。如果你对于绘制地图感兴趣，不妨下载其中的一些库看看能做出点什么。或者你可以试着连接到某些远程网络服务的 API 以免去安装软件的烦恼；第 9 章介绍了如何连接到网络服务器并解析服务器做出的 JSON 响应。

B.6.3　应用和数据

目前为止，我们已经讨论了许多关于绘制地图的内容，但其实我们可以使用地图数据做更多事情而不是仅仅停留在绘图。地理编码（Geocoding）指在地址信息和地理坐标之间进行转化。你可以找到许多与地理编码相关的 API（http://www.programmableweb.com/apitag/

geocoding），详见 ProgrammableWeb’s comparison（http://www.programmableweb.com/news/7-free-geocoding-apis-google-bing-yahoo-and-mapquest/2012/06/21）及 Python 库，例如 geopy（https://code.google.com/p/geopy/）、pygeocoder（https://pypi.python.org/pypi/pygeocoder）以及 googlemaps（http://py-googlemaps.sourceforge.net/）。如果你注册了 Google 或其他平台并获得了相应的 API 密钥，你还可以获取这些平台提供的一些其他服务，例如两位置间详细步骤的旅行路线、某一位置附近搜索，等等。

下面是一些可用的地图数据源：

- http://www.census.gov/geo/maps-data/
  美国人口统计局发布的地图文件总览
- http://www.census.gov/geo/maps-data/data/tiger.html
  大量的地理地图数据以及人口地图数据
- http://wiki.openstreetmap.org/wiki/Potential_Datasources
  全球地图资源
- http://www.naturalearthdata.com/
  三种比例尺下的矢量图以及栅格地图数据

我们还应该在这里讨论一下相关的数据科学工具（Data Science Toolkit，http://www.datasciencetoolkit.org/），包括免费的双向地理编码工具、政治版图边界的坐标信息以及统计信息，等等。你可以将所有的数据和软件统一下载到虚拟机（VM）中，从而让它们独立运行在你的电脑里不受其他软件和数据的影响。

附录C

Python的科学

在她统治期间，

蒸汽无所不能的力量被用于海陆之间，

这些都强烈地依赖于

科学上的突破。

——1887 年维多利亚女王执政 50 周年纪念颂歌，麦金泰尔 [1]

最近几年，由于大量实用软件包的出现（本章中会见到这些软件包），Python 在科学研究者中愈加流行。如果你是科学工作者或者学生，可能用过像 MATLAB 或者 R 这样的工具，也可能使用过像 Java、C、C++ 这样的传统编程语言。而本章，你将会见识 Python 为了科学研究和出版而搭建的完美平台。

C.1 标准库中的数学和统计

首先重温一下标准库，看看那些我们之前不曾提到过的模块和功能。

C.1.1 数学函数

Python 的标准 math（https://docs.python.org/3/library/math.html）库中汇集了大量的数学函数。通过 `import math` 即可将它们引入到你的程序中进行使用。

math 库会引入一些常量，例如 `pi` 和 `e`：

注 1：詹姆斯 • 麦金泰尔，加拿大诗人。

```
>>> import math
>>> math.pi
>>> 3.141592653589793
>>> math.e
2.718281828459045
```

组成 math 库的大部分内容都是函数，我们不妨先来看一些最常用的。

`fabs()` 用于获得绝对值：

```
>>> math.fabs(98.6)
98.6
>>> math.fabs(-271.1)
271.1
```

你还可以获得向下取整的整数（`floor()`）和向上取整的整数（`ceil()`）：

```
>>> math.floor(98.6)
98
>>> math.floor(-271.1)
-272
>>> math.ceil(98.6)
99
>>> math.ceil(-271.1)
-271
```

使用 `factorial()` 计算阶乘（在数学中以 n! 表示）：

```
>>> math.factorial(0)
1
>>> math.factorial(1)
1
>>> math.factorial(2)
2
>>> math.factorial(3)
6
>>> math.factorial(10)
3628800
```

使用 `log()` 计算自然对数（以 `e` 为底）：

```
>>> math.log(1.0)
0.0
>>> math.log(math.e)
1.0
```

如果你想使用别的数字作为 log 的底，可以把它当作第二个参数传入函数：

```
>>> math.log(8, 2)
3.0
```

`pow()` 做的工作与上面正好相反，它用于计算一个数的指数：

```
>>> math.pow(2, 3)
8.0
```

Python 内置的指数运算符 ** 也可以进行同样的计算，只不过当底数和指数都是整数时，

用 ** 计算得到的结果也是整数，不会被自动转化为浮点数：

```
>>> 2**3
8
>>> 2.0**3
8.0
```

使用 sqrt() 得到平方根：

```
>>> math.sqrt(100.0)
10.0
```

别想着调戏 sqrt() 函数，你的小把戏它都见过：

```
>>> math.sqrt(-100.0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: math domain error
```

常见的三角函数都可以使用，这里我只列出它们的名字：sin()、cos()、tan()、asin()、acos()、atan()、atan2()。如果你还记得勾股定理，math 库还包含 hypot()[2] 函数帮你计算给定两直角边对应的斜边长度：

```
>>> x = 3.0
>>> y = 4.0
>>> math.hypot(x, y)
5.0
```

如果你不放心这些神奇的函数，可以自己验证一下，看看它们是否能正常工作：

```
>>> math.sqrt(x*x + y*y)
5.0
>>> math.sqrt(x**2 + y**2)
5.0
```

最后一组函数用于进行角坐标转换：

```
>>> math.radians(180.0)
3.141592653589793
>>> math.degrees(math.pi)
180.0
```

C.1.2 使用复数

基本 Python 语言（即不引入任何第三方库的 Python）就支持复数运算。所谓复数，就是由实部（real）和虚部（imaginary）两部分组成的数：

```
>>> # 一个实数
... 5
5
>>> # 一个虚数
... 8j
```

注 2：函数名 hypot 是 hypotenuse 的缩写，意思是斜边。——译者注

```
8j
>>> # 一个虚数
... 3 + 2j
(3+2j)
```

由于虚数 `i`（在 Python 中用 `1j` 表示）被定义为 −1 的平方根，因此我们可以执行下列运算：

```
>>> 1j * 1j
(-1+0j)
>>> (7 + 1j) * 1j
(-1+7j)
```

一些与复数计算相关的函数都被纳入了 `cmath`（https://docs.python.org/3/library/cmath.html）模块。

C.1.3　使用小数对浮点数进行精确计算

计算机中的浮点数和我们在学校学的实数有些差别。由于计算机的 CPU 是针对二进制运算设计的，因此它无法准确表示那些不是 2 的幂次方的数：

```
>>> x = 10.0 / 3.0
>>> x
3.3333333333333335
```

天呐，最后的那个 5 是怎么回事？应该一直是 3 才对啊。使用 Python 的 `decimal`（https://docs.python.org/3/library/decimal.html）模块，你可以把数字按任意你所需的精度表示。这种对于涉及钱的计算非常重要。美元的最小精度是 1 美分（1 美元的百分之一），因此当我们需要用美元和美分计算钱的数目时，我们期望最终的结果精确到美分。如果我们使用像 19.99 和 0.06 这样的浮点数来表示美元和美分的话，在开始实际计算前，我们的数据精度就会有所损失（浮点数的最后几位数字会像上面的例子一样产生误差）。那么如何解决这个问题呢？很简单，使用 `decimal` 模块代替即可：

```
>>> from decimal import Decimal
>>> price = Decimal('19.99')
>>> tax = Decimal('0.06')
>>> total = price + (price * tax)
>>> total
Decimal('21.1894')
```

我们使用字符串值初始化了价格（price）和税收（tax）变量，以此隐性指明数据的有效数字。`total` 的计算过程保留了所有有效数字，但是我们只需要精确到最近的美分：

```
>>> penny = Decimal('0.01')
>>> total.quantize(penny)
Decimal('21.19')
```

上例中，也许你使用浮点数和取整运算能获得一样的结果，但这并不总能成功。你也可以把所有数都乘以 100，然后使用整数进行运算，但小心，这早晚也会产生问题。在 www.itmabyahack.com（http://www.itmaybeahack.com/homepage/books/nonprog/html/p13_modules/p13_c03_decimal.html）上，有许多关于上述计算精度损失问题的讨论。

C.1.4 使用分数进行有理数运算

你可以使用标准 Python 中的 `fractions`（https://docs.python.org/3/library/fractions.html）模块将有理数表示为分子除以分母的形式。下面是一个简单的例子，计算三分之一乘以三分之二：

```
>>> from fractions import Fraction
>>> Fraction(1,  3) * Fraction(2, 3)
Fraction(2, 9)
```

使用浮点数作为参数会造成结果不准确，你可以在 `Fraction` 中使用 `Decimal` 作为替代传入：

```
>>> Fraction(1.0/3.0)
Fraction(6004799503160661, 18014398509481984)
>>> Fraction(Decimal('1.0')/Decimal('3.0'))
Fraction(3333333333333333333333333333, 10000000000000000000000000000)
```

使用 `gcd` 函数可以获得两个数的最大公约数：

```
>>> import fractions
>>> fractions.gcd(24, 16)
8
```

C.1.5 使用`array`创建压缩序列

Python 中的列表类型更像是一个链表而不是数组。如果你需要一个元素类型全都相同的一维序列，可以使用数组（`array`，https://docs.python.org/3/library/array.html）类型。与列表相比，数组占用空间更小，却支持许多列表方法。使用 `array(typecode, initializer)` 可以创建一个新的数组。其中 typecode 定义了数据类型（例如 `int` 或 `float`），另一个可选的参数 `initializer` 包含了初始化的值，它可以是一个列表、字符串或者其他可迭代类型。

在实际工作中，我从未使用过这个包。数组是一个相对低层次的数据结构，在表示图像数据时比较有用。如果你使用数组——尤其是多维数组——仅仅是为了进行数值计算的话（例如矩阵运算），最好还是使用 NumPy 代替，随后就会讲到这个包。

C.1.6 使用`statistics`进行简单数据统计

从 Python 3.4 开始，`statistics`（https://docs.python.org/3.4/library/statistics.html）被纳入了标准模块。它包含许多实用函数：平均值、中数、求模、标准差、方差，等等。输入的数据可以是序列型的（列表或元组）也可以是不同类型的数值（整数、浮点数、小数、分数）组成的迭代型。其中，`mode` 函数还可以接受字符串类型的输入。SciPy 和 Pandas 包含更多的统计函数，随后会介绍。

C.1.7 矩阵乘法

据说从 Python 3.5 开始，`@` 字符将不再仅限于处理字符时使用了。它仍然还会具有修饰符的作用，但同时还将可以用于*矩阵乘法*（http://legacy.python.org/dev/peps/pep-0465/）。但在

它真正到来之前，NumPy 仍然是你进行矩阵运算的最佳选择。

C.2　科学Python

本附录剩下的内容会涵盖一些与科学、数学相关的第三方 Python 包。你可以选择分开安装它们，但我建议通过安装某个科学计算版本的 Python 从而一口气把它们全都安装了。下面是几种主流的科学版 Python。

- Anaconda（https://store.continuum.io/cshop/anaconda/）
  这个包是免费的、可扩展的、不断更新的科学计算版 Python。它同时支持 Python 2 和 Python 3，而且不会对你系统中已经安装的 Python 产生任何副作用。
- Enthought Canopy（https://www.enthought.com/products/canopy/）
  这个包既有免费版本也有商业版本。
- Python(*x*,*y*)（https://code.google.com/p/pythonxy/）
  这个包仅可在 Windows 下使用。
- Pyzo（http://www.pyzo.org/）
  这个包是建立在 Anaconda 的一些工具上的，同时添加了一些其他的实用工具。
- ALGORETE Loopy（http://algorete.org/）
  这也是一个建立在 Anaconda 上的包，同样添加了一些其他内容。

我建议安装 Anaconda。虽然它的体积比较庞大，但包含了本附录中会使用到的所有工具，一劳永逸。你可以参考附录 D，学习如何安装 Python 3 版本的 Anaconda。本附录的后半部分内容会假定你已经安装好了所有所需的包，不管你是分开安装还是通过 Anaconda 安装。

C.3　Numpy

NumPy（http://www.numpy.org/）是 Python 在科学工作者中流行起来的主要原因之一。你可能听到过类似这样的说法：像 Python 这样的动态语言要比像 C 语言一样的编译型语言效率低，甚至比像 Java 这样的解释型语言的效率也要低。NumPy 是为了进行快速多维矩阵运算而设计的，和 FORTRAN 这样的科学性语言有些类似。你可以同时获得接近 C 语言的运算速度和 Python 的友好（便于编写使用）接口。

如果你已经下载安装了 Python 的某一个科学计算版本，应该已经拥有 NumPy 了。如果没有的话可以前往 NumPy 的下载页（http://www.scipy.org/scipylib/download.html）按教程下载安装。

在开始使用 NumPy 之前，你需要理解它的核心数据结构：多维数组，用 `ndarray`（N-dimensional array）或仅仅 `array` 表示。和 Python 的列表、元组不同，多维数组中的元素必须是同一类型的。NumPy 把数组的维数称为它的*秩*（rank）。一维数组就像是一行数据，二维数组就像是一张包含行列的表格，三维数组就像是一个魔方。每一维的长度不要求相同。

> NumPy 中的 array 和标准 Python 中的 array 并不是一回事。本附录的后面部分，当我说数组时，我指的都是 NumPy 里面的数组。

为什么需要数组类型呢？

- 科学数据总是包含大量序列数据；
- 对这种数据进行的科学计算经常包括矩阵数学、回归运算、模拟以及其他需要同时对许多数据点进行运算的操作；
- NumPy 对数组的处理速度要远远超过它处理标准 Python 里列表或元组的速度。

NumPy 数组有多种创建方式。

C.3.1　使用array()创建数组

你可以利用普通的列表或元组生成数组：

```
>>> b = np.array( [2, 4, 6, 8] )
>>> b
array([2, 4, 6, 8])
```

ndim 属性会返回数组的秩：

```
>>> b.ndim
1
```

数组中所有元素的个数由 size 记录：

```
>>> b.size
4
```

每一秩（维）的元素数量由 shape 返回：

```
>>> b.shape
(4,)
```

C.3.2　使用arange()创建数组

NumPy 中的 arrange() 与 Python 中的标准 range() 类似。如果你在调用 arange() 时只传入了一个整形参数 num，它会返回一个包含了从 0 到 num-1 的整数的 ndarray：

```
>>> import numpy as np
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a.ndim
1
>>> a.shape
(10,)
>>> a.size
10
```

如果传入两个参数，则 arange() 方法会返回从第一个参数到最后一个参数减一的全部整数的数组：

```
>>> a = np.arange(7, 11)
>>> a
array([ 7,  8,  9, 10])
```

除此之外，你还可以把自己指定的步长（默认是 1）作为第三个参数：

```
>>> a = np.arange(7, 11, 2)
>>> a
array([7, 9])
```

目前为止，我们的例子中出现的都是整数，但事实上，使用浮点数也能完美运行：

```
>>> f = np.arange(2.0, 9.8, 0.3)
>>> f
array([ 2. ,  2.3,  2.6,  2.9,  3.2,  3.5,  3.8,  4.1,  4.4,  4.7,  5. ,
        5.3,  5.6,  5.9,  6.2,  6.5,  6.8,  7.1,  7.4,  7.7,  8. ,  8.3,
        8.6,  8.9,  9.2,  9.5,  9.8])
```

最后一个小技巧，dtype 参数可以指定 arange 产生的值的类型：

```
>>> g = np.arange(10, 4, -1.5, dtype=np.float)
>>> g
array([ 10. ,   8.5,   7. ,   5.5])
```

C.3.3　使用zeros()、ones()和random()创建数组

zeros() 会返回一个全都是 0 的矩阵。zeros 接收的参数是元组，用于指定你想要创建的矩阵的形状。下面是一个一维数组的例子：

```
>>> a = np.zeros((3,))
>>> a
array([ 0.,  0.,  0.])
>>> a.ndim
1
>>> a.shape
(3,)
>>> a.size
3
```

下面创建的矩阵的秩为二：

```
>>> b = np.zeros((2, 4))
>>> b
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]])
>>> b.ndim
2
>>> b.shape
(2, 4)
>>> b.size
8
```

另一个特殊的函数 ones()，它创建的矩阵中所有值都相等：

```
>>> import numpy as np
>>> k = np.ones((3, 5))
>>> k
array([[ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.,  1.]])
```

最后一个初始化函数可以创建由 0.0 到 1.0 之间的随机数组成的矩阵：

```
>>> m = np.random.random((3, 5))
>>> m
array([[  1.92415699e-01,   4.43131404e-01,   7.99226773e-01,
          1.14301942e-01,   2.85383430e-04],
       [  6.53705749e-01,   7.48034559e-01,   4.49463241e-01,
          4.87906915e-01,   9.34341118e-01],
       [  9.47575562e-01,   2.21152583e-01,   2.49031209e-01,
          3.46190961e-01,   8.94842676e-01]])
```

C.3.4 使用reshape()改变矩阵的形状

目前为止，你可能还是觉得数组能做的事情和列表、元组没有什么区别。其实，数组能做许多列表和元组做不到的事情，例如你可以使用 reshape() 函数对数组的形状进行修改：

```
>>> a = np.arange(10)
>>> a
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
>>> a = a.reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a.ndim
2
>>> a.shape
(2, 5)
>>> a.size
10
```

你可以将刚才的数组转化为其他形状：

```
>>> a = a.reshape(5, 2)
>>> a
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> a.ndim
2
>>> a.shape
(5, 2)
>>> a.size
10
```

将表示形状的元组赋值给 shape 属性也可以达到相同的效果：

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

形状的唯一限制是秩的乘积需要等于数组中元素的个数（上例中是 10）：

```
>>> a = a.reshape(3, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: total size of new array must be unchanged
```

C.3.5 使用[]访问元素

一维数组的使用和列表类似：

```
>>> a = np.arange(10)
>>> a[7]
7
>>> a[-1]
9
```

然而，如果数组的形状不是一维的，就需要用逗号分开指定每一维的索引了：

```
>>> a.shape = (2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
>>> a[1,2]
7
```

这和二维的 Python 列表的访问方法有所不同：

```
>>> l = [ [0, 1, 2, 3, 4], [5, 6, 7, 8, 9] ]
>>> l
[[0, 1, 2, 3, 4], [5, 6, 7, 8, 9]]
>>> l[1,2]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list indices must be integers, not tuple
>>> l[1][2]
7
```

还有一件事情。分片操作仍然可用，只不过你需要将它们全都放到一组方括号中。再次创建我们熟悉的测试数组：

```
>>> a = np.arange(10)
>>> a = a.reshape(2, 5)
>>> a
array([[0, 1, 2, 3, 4],
       [5, 6, 7, 8, 9]])
```

使用分片操作访问第一行中从偏移量为 2 的元素开始的所有元素：

```
>>> a[0, 2:]
array([2, 3, 4])
```

接下来试着获取最后一行的倒数三个元素：

```
>>> a[-1, :3]
array([5, 6, 7])
```

你还可以使用分片同时给多个元素赋值。例如下面的语句将每一行的第 2 列和第 3 列（偏移位置）元素赋值为 1000：

```
>>> a[:, 2:4] = 1000
>>> a
array([[   0,    1, 1000, 1000,    4],
       [   5,    6, 1000, 1000,    9]])
```

C.3.6　数组运算

创建数组和数组变形太有趣了，以至于我们差点忘了介绍如何对数组中的数据进行处理。下面要介绍的第一个技巧是使用 NumPy 中重定义的乘法运算符（*）对 NumPy 数组中的所有元素进行批量乘法：

```
>>> from numpy import *
>>> a = arange(4)
>>> a
array([0, 1, 2, 3])
>>> a *= 3
>>> a
array([0, 3, 6, 9])
```

如果你想要对一个普通 Python 列表中所有元素进行乘法操作，需要使用循环或者列表迭代：

```
>>> plain_list = list(range(4))
>>> plain_list
[0, 1, 2, 3]
>>> plain_list = [num * 3 for num in plain_list]
>>> plain_list
[0, 3, 6, 9]
```

这种批量操作同样适用于加法、减法、除法以及其他一些 NumPy 库中的函数。例如，通过组合使用 zeros() 和 +，你可以将新创建的数组中所有元素初始化为某一个相同的值：

```
>>> from numpy import *
>>> a = zeros((2, 5)) + 17.0
>>> a
array([[ 17.,  17.,  17.,  17.,  17.],
       [ 17.,  17.,  17.,  17.,  17.]])
```

C.3.7　线性代数

NumPy 包含许多与线性代数有关的函数，比如我们定义下面这组线性方程：

```
4x + 5y = 20
 x + 2y = 13
```

如何对 x 和 y 进行求解？首先需要创建两个矩阵：

- 系数矩阵（coefficient，x 和 y 的系数组成的矩阵）
- 因变量矩阵（等式右侧）

```
>>> import numpy as np
>>> coefficients = np.array([ [4, 5], [1, 2] ])
>>> dependents = np.array( [20, 13] )
```

现在使用 `linalg` 模块中的 `solve()` 函数：

```
>>> answers = np.linalg.solve(coefficients, dependents)
>>> answers
array([ -8.33333333,  10.66666667])
```

运行结果显示 x 的值大约是 −8.3，y 的值大约是 10.6。这个解正确吗？

```
>>> 4 * answers[0] + 5 * answers[1]
20.0
>>> 1 * answers[0] + 2 * answers[1]
13.0
```

觉得有些繁琐？那看看下面的做法如何。直接让 NumPy 为我们计算两个矩阵的点积（dot product），以省去一些敲打键盘的时间：

```
>>> product = np.dot(coefficients, answers)
>>> product
array([ 20.,  13.])
```

如果得到的解是正确的，那么 `product` 数组中的值应该与 `dependents` 中的值相近。你可以使用 `allclose()` 函数来检查两个数组是否近似相等（点积得到的结果可能与因变量矩阵不完全相同，因为存在浮点小数的取整问题）：

```
>>> np.allclose(product, dependents)
True
```

NumPy 还提供了与多项式、傅立叶变换、统计以及一些概率分布相关的模块。

C.4　SciPy库

SciPy（http://www.scipy.org/）是一个建立在 NumPy 的基础之上的库，它包含了更多的数学函数和统计函数。SciPy 发布包（http://www.scipy.org/scipylib/download.html）包括 NumPy、SciPy、Pandas（本附录后面会讲到）以及一些其他的库。

SciPy 包含许多模块，可以完成下面这些计算任务：

- 优化
- 统计
- 插值

- 线性回归
- 积分
- 图像处理
- 信号处理

如果你曾经使用过其他的科学计算工具，就会发现 Python、NumPy、SciPy 可以涵盖商业的 MATLAB（http://cn.mathworks.com/products/matlab/）以及开源的 R（http://www.r-project.org/）中的大部分功能。

C.5 SciKit库

同样遵循根据已有的软件包进行扩展的模式，SciKit（https://scikits.appspot.com/scikits）是一组建立在 SciPy 基础上的科学计算软件包。SciKit 的特长在于机器学习（machine learning）。它支持建模、分类、聚类以及多种机器学习算法。

C.6 IPython库

IPython（http://ipython.org/）的诸多优点让它非常值得我们花时间学习，下面列出了其中几点：

- 改进的交互式解释器（简单来说，就是本书中一直使用的 >>> 的扩展）；
- 在基于网页的笔记本（notebook）中发布代码、图形、文本以及其他形式的文件；
- 支持并行化计算（parallel computing，http://ipython.org/ipython-doc/stable/parallel/parallel_intro.html）。

我们主要来看看解释器和笔记本部分。

C.6.1 更好的解释器

IPython 有分别针对 Python 2 和 Python 3 的两个不同的版本，在安装 Anaconda 或其他科学计算版 Python 时会自动安装。使用 `ipython3` 运行与 Python 3 对应的版本。

```
$ ipython3

 Python 3.3.3 (v3.3.3:c3896275c0f6, Nov 16 2013, 23:39:35)
 Type "copyright", "credits" or "license" for more information.

 IPython 0.13.1 -- An enhanced Interactive Python.
 ?         -> Introduction and overview of IPython's features.
 %quickref -> Quick reference.
 help      -> Python's own help system.
 object?   -> Details about 'object', use 'object??' for extra details.

 In [1]:
```

标准 Python 解释器使用 >>> 和 `...` 作为输入提示符，其中，`...` 表示你仍需要输入一些代码以保证正常运行。IPython 会将你输入的所有东西保存在名为 `In` 的列表中，同时将所有

输入的内容保存到 Out 列表中。每个输入可以包含多行数据，通过按下 Shift 加回车可以结束输入并提交你输入的内容。下面是一个一行的例子：

```
In [1]: print("Hello? World?")
Hello? World?

In [2]:
```

In 和 Out 是自动编号的列表，这让你可以方便地访问任何你输入的内容和得到的输出内容。

如果你在一个变量后边输入 ?，IPython 会显示出它的类型、值、创建该类变量的方式以及其他一些相关信息：

```
In [4]: answer = 42

In [5]: answer?

Type:       int
String Form:42
Docstring:
int(x=0) -> integer
int(x, base=10) -> integer

Convert a number or string to an integer, or return 0 if no arguments
are given.  If x is a number, return x.__int__().  For floating point
numbers, this truncates towards zero.

If x is not a number or if base is given, then x must be a string,
bytes, or bytearray instance representing an integer literal in the
given base.  The literal can be preceded by '+' or '-' and be surrounded
by whitespace.  The base defaults to 10.  Valid bases are 0 and 2-36.
Base 0 means to interpret the base from the string as an integer literal.
>>> int('0b100', base=0)
4
```

名称查找是像 IPython 这样的 IDE 中常见的功能。如果你在输入了一串字符后按下 Tab 键，IPython 会显示出所有以当前字符串为开头的变量、关键词、函数，等等。我们首先来定义一些变量，然后尝试查找所有以 f 开头的内容：

```
In [6]: fee = 1

In [7]: fie = 2

In [8]: fo = 3

In [9]: fum = 4

In [10]: ftab
%%file    fie       finally   fo        format    frozenset
fee       filter    float     for       from      fum
```

如果你在输入 fe 后按下 Tab 键，它会自动扩展为变量 fee，因为它是当前程序中唯一一个以 fe 开头的内容：

```
In [11]: fee
Out[11]: 1
```

C.6.2　IPython笔记本

如果你倾向于图形界面，可能会喜欢 IPython 的基于网页的实现。首先打开 Anaconda 的开始窗口（图 C-1）。

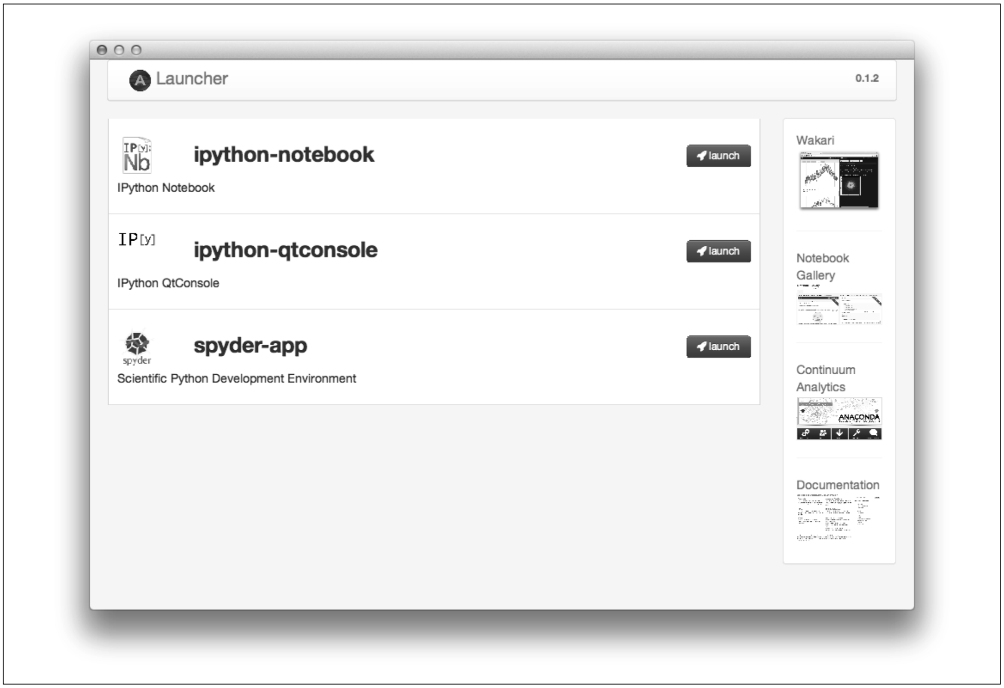


图 C-1：Anaconda 开始界面

点击 ipython-notebook 右侧的 launch 图标即可在浏览器中运行笔记本。图 C-2 显示了初始启动页。



图 C-2：iPython 主界面

现在点击创建新笔记本会弹出一个类似图 C-3 的窗口。

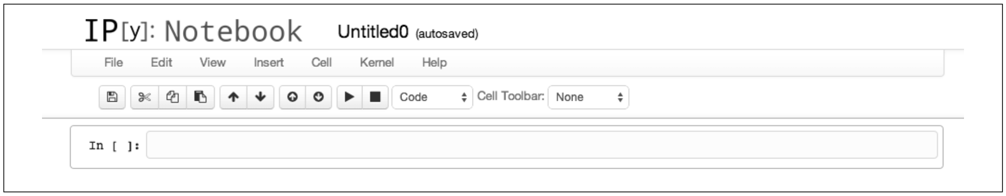


图 C-3：iPython 笔记本页面

在图形界面下重新输入我们之前例子中的代码，如图 C-4 所示。

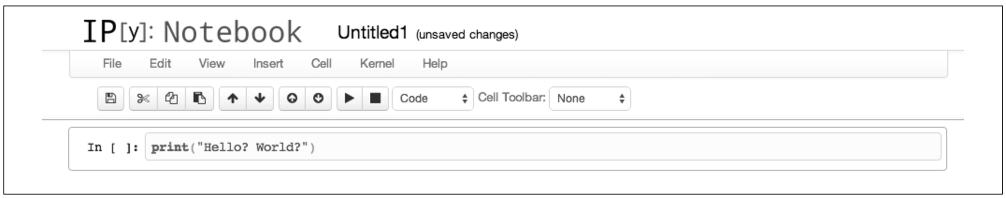


图 C-4：在 iPython 中输入代码

点击黑色的小三角运行它。运行结果如图 C-5 所示。

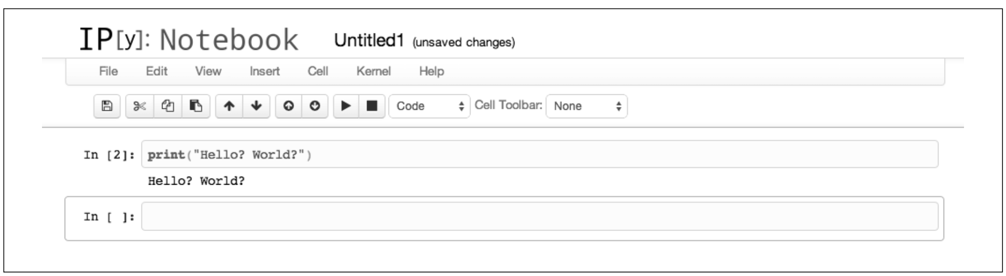


图 C-5：在 iPython 中运行代码

IPython 的笔记本不仅仅是图形化版的 IPython 解释器。除了代码，它还可以包含文本、图片、格式化的数学表达式，等等。

在笔记本界面上方的一排按钮中有一个是下拉菜单（图 C-6），它可以指定你输入内容的方式。下面是几种可选的方式。

- Code
  默认的输入方式，用于输入 Python 代码。
- Markdown
  HTML 的一种替代，可以将输入文本按照预先指定的格式处理为可读文本。
- 纯文本
  无格式文本，从 Heading 1 到 Heading 6，即对应 HTML 中的 <H1> 到 <H6> 标签。

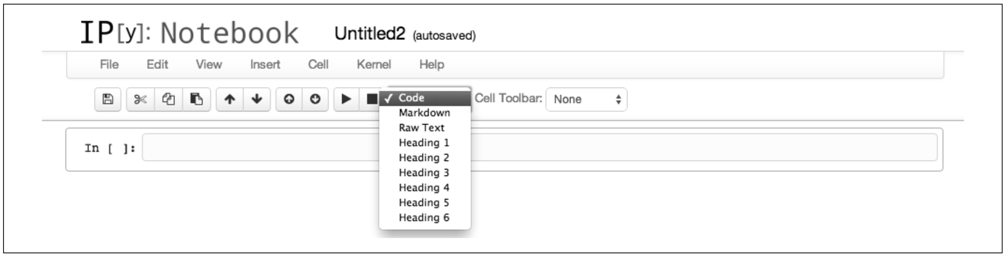


图 C-6：可选内容格式菜单

试着在我们的代码中穿插加入些文本，让它看起来像是某种 wiki 页。从下拉菜单中选择 Heading 1 然后输入“Humble Brag Example”，接着按下 Shift 加回车键（Enter）。你应该能看到粗体显示的上面三个单词。然后从下拉菜单中选择 Code 并随便输入一些代码，比如下面这些：

```
print("Some people say this code is ingenious")
```

再次按下 Shift+Enter 来完成输入。现在你应该能看到格式化的标题和代码，如图 C-7 所示。



图 C-7：格式化文本和代码

通过将输入代码、输出结果、文本甚至图片穿插起来，你可以创建交互式的笔记本。又由于它是基于网页的，因此你可以从任何浏览器上访问到，十分便捷。

你可以看到一些转化成静态 HTML（http://nbviewer.ipython.org/）的笔记，或者放置在收藏（https://github.com/ipython/ipython/wiki/A-gallery-of-interesting-IPython-Notebooks）中的笔记。举个具体的例子，你可以看看关于泰坦尼克号上乘客的笔记（http://nbviewer.ipython.org/github/agconti/kaggle-titanic/blob/master/Titanic.ipynb）。它包括一些图表，展示了性别、穷富、位于船上的位置等因素是如何影响生存率的。作为奖励，你还可以读一读如何使用不同的机器学习方法。

科学家们也开始使用 IPython 笔记来发表他们的研究成果，包括所有代码和支持结论的数据。

C.7 Pandas

最近几年，*数据科学*（data science）这个词变得越来越流行。关于这个词，我看到过这样一些解释："在 Mac 上完成的统计"或"在旧金山完成的统计"。然而，不管你如何定义数据科学的范畴，本章我们讨论的工具——NumPy、SciPy 以及这一节的主角 Pandas——都是日益流行的数据科学工具中的重要成员。（当然 Mac 和旧金山都是可有可无的。）

Pandas（http://pandas.pydata.org/）是一款全新的用于进行交互式数据分析的工具包。Pandas 融合了 NumPy 进行矩阵运算的能力以及处理电子表格和关系型数据库的能力，因此它在处理真实世界的数据上非常有效。Wes McKinney 编写的 *Python for Data Analysis: Data Wrangling with Pandas, NumPy, and IPython*（O'Reilly，http://shop.oreilly.com/product/0636920023784.do）讲解了如何使用 NumPy、IPython 以及 Pandas 进行有效的数据处理。

NumPy 是针对传统科学计算而设计的，它主要用于处理单一类型的多维数据集，通常为浮点型数据。Pandas 更像是一个数据库编辑器，可以处理包含多种数据类型的组（group）。在某些语言中，这种组也被称为*记录*（record）或者*结构*（structure）。Pandas 定义一种基本数据结构，叫作 `DataFrame`（数据帧）。它是一个有序的列的集合，每一列都有自己的名称和类型。数据帧有点像数据库中的表格、Python 中的命名元组和嵌套字典。Pandas 的设计目的在于简化不同类型数据的处理过程，使它不仅仅适用于科学计算中遇到的类型（主要是浮点型）还适用于商业数据。事实上，Pandas 最初就是为了处理金融数据而设计的，它最常见的替代品就是电子表格。

Pandas 是一款用于处理真实世界中各种类型的混杂数据（值的缺失、古怪的格式、稀疏的测量值，等等）的 ETL 工具。你可以用它对数据文件进行划分（split）、合并（join）、扩展（extend）、添加（fill in）、转换（convert）、变形（reshape）、切分（slice）、加载（load）和保存（save）等操作。它整合了我们之前讨论过的那些工具——NumPy、SciPy、IPython——用于进行统计计算、根据模型拟合数据、绘制图表、发表，等等。

大多数科学家们都希望能尽快把他们的工作做完，而不需要花费几个月的时间先成为某种晦涩的计算机语言或应用程序的专家。有了 Python，他们很快就能变得非常有效率。

C.8 Python和科学领域

之前我们讨论的都是可以应用到任何科学领域的 Python 工具。那么，有没有专门针对于某一特定领域的 Python 工具和相关文档呢？下面列出了 Python 在某些特定问题方向的应用，以及一些适用于特定问题的 Python 库。

- 通用
    - 在科学和工程中使用 Python 进行计算（http://kitchingroup.cheme.cmu.edu/pycse/pycse.html）
    - 科学家的 Python 速成课（http://nbviewer.ipython.org/gist/rpmuller/5920182）

- 物理
  - 计算物理学（http://www-personal.umich.edu/~mejn/computational-physics/）
- 生物和医药
  - 生物学家应学的 Python（http://pythonforbiologists.com/）
  - 使用 Python 处理神经影像（http://nipy.org/）

下面列出一些与 Python 和科学数据处理相关的国际会议：

- PyData（http://pydata.org/）
- SciPy（http://conference.scipy.org/）
- EuroSciPy（https://www.euroscipy.org/）

附录D

安装Python 3

等到 Python 3 被预装到每一台计算机上时，估计烤面包机都已经被可以打印酥粒甜甜圈的 3D 打印机取代了。Windows 压根没有 Python，OS X、Linux、Unix 也都只是预装了旧版本的 Python 而已。在它们赶上 Python 最新版本前，我们只能自己手动安装 Python 3。

下面几节详细描述了如何做到以下几件事情：

- 查询你电脑中安装的 Python 版本（如果安装了）
- 安装标准发布版的 Python 3（如果没有安装）
- 安装 Anaconda 发布的 Python 科学计算模块
- 如果无法直接修改你的系统（权限问题等），不妨安装 `pip` 和 `virtualenv` 进行第三方包的管理
- 安装 `conda` 作为 `pip` 的替代品

本书中大部分例子都是在 Python 3.3 的环境下编写并测试的，这是写本书时最新的稳定版本。有些是在 3.4 下编写并测试的，这个版本是在校对本书时发布的。新版本 Python 的变化网页（https://docs.python.org/3/whatsnew/）里描述了每一个新版本更新的内容。Python 安装包有多种来源，安装新版本的方式也不止一种。本附录会介绍其中两种安装方式。

- 如果你只会用标准解释器和库，建议从官方语言网站（https://www.python.org/）下载安装。
- 如果你除了使用 Python 的标准库外还需要使用一些附录 C 中描述的强大的科学计算库，就下载安装 Anaconda 吧。

D.1　安装标准Python

打开浏览器前往 Python 下载页（https://www.python.org/downloads/）。该网页会尝试获取你

的操作系统类型并显示对应的 Python 版本。如果它显示的不对，你可以手动从下面几个链接中选择对应的版本下载：

- Python Releases for Windows（https://www.python.org/downloads/windows/）
- Python Releases for Mac OS X（https://www.python.org/downloads/mac-osx/）
- Python Source Releases（Linux and Unix，https://www.python.org/downloads/source/）

你会看到类似图 D-1 的网页。

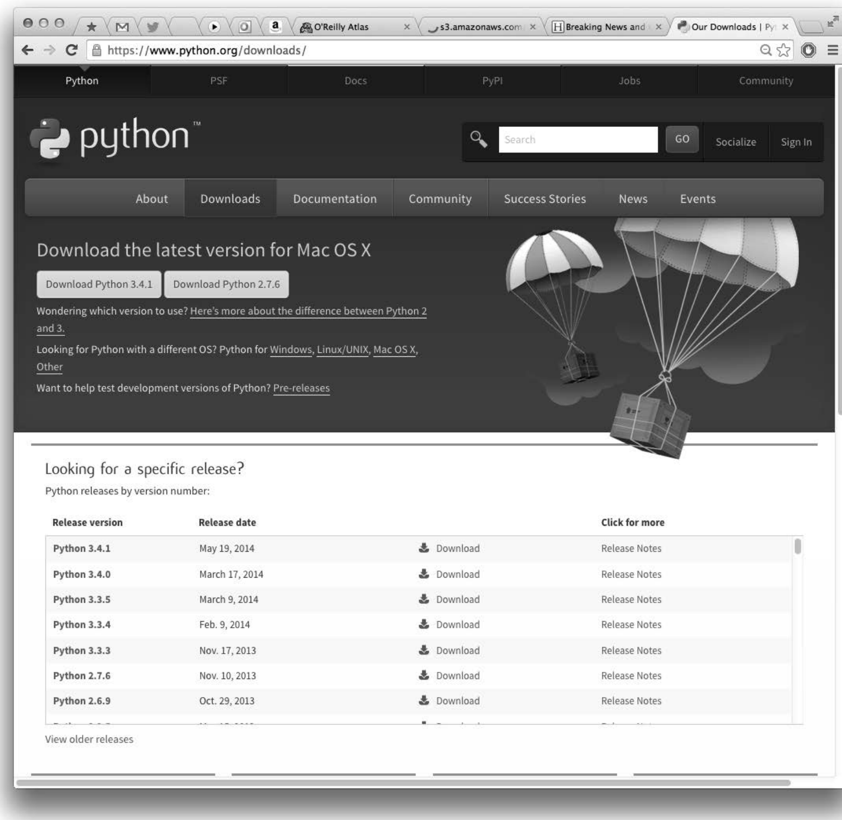


图 D-1：下载页实例

点击最新版本的下载链接。在编写本书时是 3.4.1。点击后会跳转到详细信息页，如图 D-2 所示。

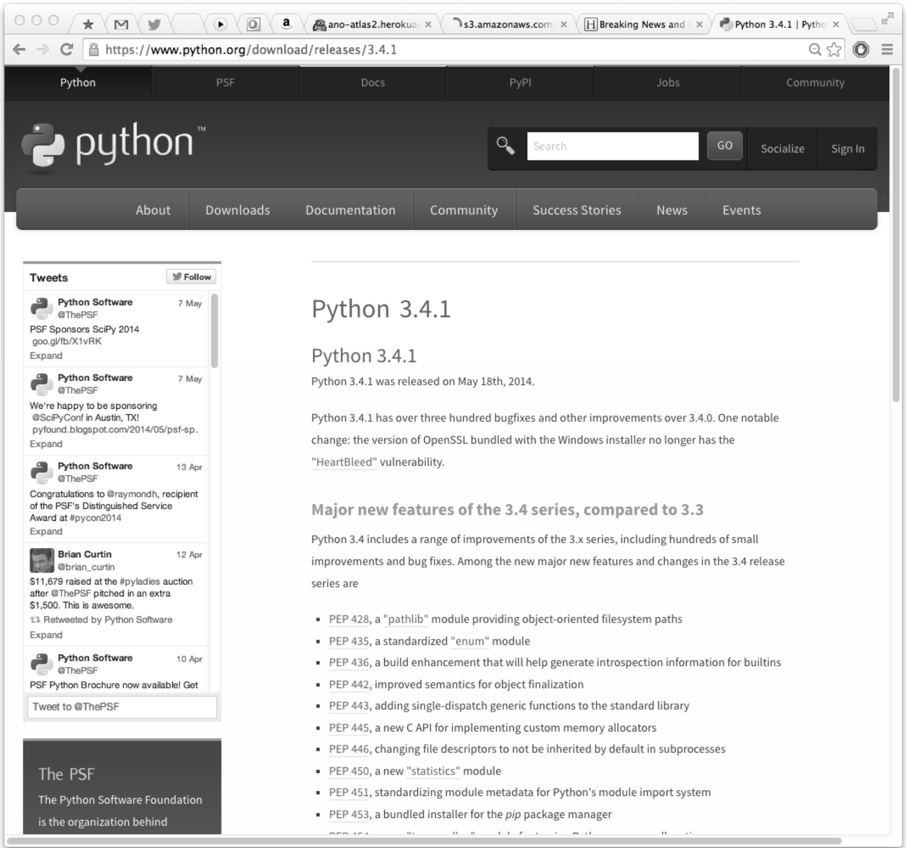


图 D-2：下载详情页

你需要向下滚动才能找到实际下载链接（如图 D-3 所示）。

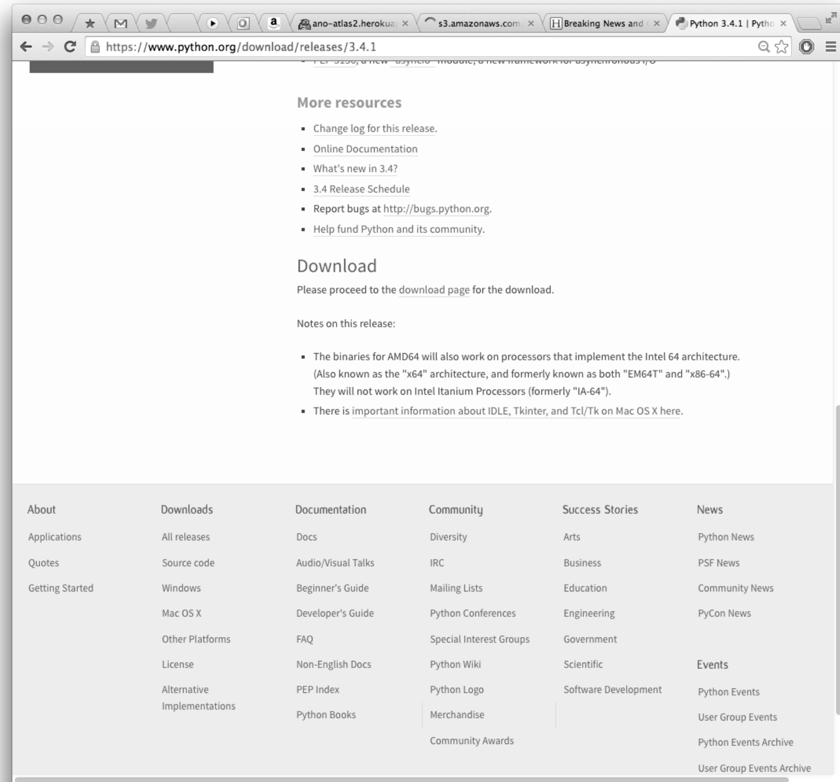


图 D-3：下载链接页的底部

点击下载链接可以跳转到实际发布版本页（如图 D-4 所示）。

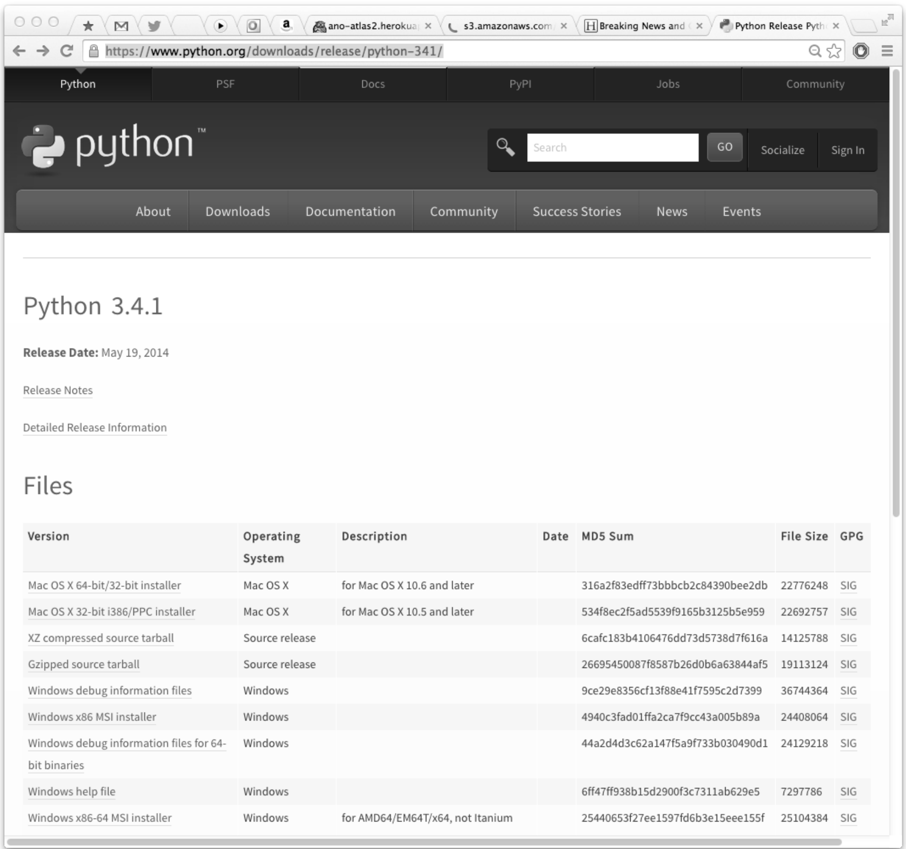


图 D-4：可以下载的文件

现在点击对应版本的链接即可将 Python 下载到你的电脑了。

D.1.1　Mac OS X

点击 Mac OS X 64-bit/32-bit installer（https://www.python.org/ftp/python/3.4.1/python-3.4.1-macosx10.6.dmg）链接下载 Mac 中使用的 .dmg 文件。下载完成后双击它，桌面上会弹出一个包含 4 个图标的窗口。右键单击 Python.mpkg，在弹出的对话框中点击“打开”。点击大约三次“继续”按钮，期间会显示一些法律声明，最后的对话框出现后点击“安装”。Python 3 的所有内容都会被安装到 /usr/local/bin/python3，不会对电脑上已安装的 Python 2 产生任何影响。

D.1.2　Windows

对于 Windows，根据自己计算机的具体环境选择下面两个安装包之一：

- Windows x86 MSI installer（32 位，https://www.python.org/ftp/python/3.4.1/python-3.4.1.msi）
- Windows x86-64 MSI installer（64 位，https://www.python.org/ftp/python/3.4.1/python-3.4.1.amd64.msi）

如果你不清楚自己的 Windows 到底是 32 位还是 64 位，可以执行下面几步来查看：

(1) 点击“开始”按钮；
(2) 右键点击“计算机”；
(3) 点击“属性”，就可以找到相关信息了。

点击适当的安装包链接（.msi 文件）。下载完成后双击打开，并根据安装指导进行安装。

D.1.3 Linux或Unix

Linux 和 Unix 用户可以选择下载 Python 源代码的压缩文件：

- XZ compressed source tarball（https://www.python.org/ftp/python/3.4.1/Python-3.4.1.tar.xz）
- Gzipped source tarball（https://www.python.org/ftp/python/3.4.1/Python-3.4.1.tgz）

下载上面两个链接中的一个。使用 `tar xJ`（.xz 文件）或 `tar xz`（.tgz 文件）命令解压，然后运行解压得到的 shell 脚本即可。

D.2 安装Anaconda

Anaconda 是一个针对科学计算设计的多合一安装包：包括 Python 本体、标准库以及许多实用的第三方库。目前为止，Anaconda 自带的仍是 Python 2 以及对应的解释器，但我们可以使用一些变通的方法强制它安装 Python 3。

最新的升级版本，Anaconda 2.0 会安装最近版本的 Python 及其标准库（编写此书时是 3.4）。它还会安装一些其他的实用工具，包括本书中讨论过的一些库：`beautifulsoup4`、`flask`、`ipython`、`matplotlib`、`nose`、`numpy`、`pandas`、`pillow`、`pip`、`scipy`、`tables`、`zmq`，等等。Anaconda 还包括一个跨平台的安装程序 `conda`，它是 `pip` 的增强版，之后会简单介绍它。

前往与 Python 3 版本对应的下载页（http://repo.continuum.io/anaconda3/）下载安装 Anaconda 2。点击与你使用的平台对应的链接（具体的版本信息可能与下面列出的不一致，但我相信你能识别出应该下载哪一个）。

- 如果需要在 Mac 上使用，点击 Anaconda3-2.0.0-MacOSX-x86_64.pkg（http://repo.continuum.io/anaconda3/Anaconda3-2.0.0-MacOSX-x86_64.pkg）。“下载完成后双击打开文件，然后按提示步骤安装 Anaconda 即可。安装器会将所有内容安装到 home 目录[1]下面的 anaconda 子目录中。
- 对于 Windows 用户，点击 32-bit version（http://bit.ly/a/warning?url=http%3a%2f%2frepo

注 1：一般是 /Users/Your_Name。——译者注

%2econtinuum%2eio%2fanaconda3%2fAnaconda3%2d2%2e0%2e0%2dWindows%2dx86%2eexe&hash=win-32b）或者 64-bit version（http://bit.ly/a/warning?url=http%3a%2f%2frepo%2econtinuum%2eio%2fanaconda3%2fAnaconda3%2d2%2e0%2e0%2dWindows%2dx86%5f64%2eexe&hash=win-64b）进行下载。下载完成后双击 .exe 进行安装。

- 对于 Linux 用户，点击 32-bit version（http://repo.continuum.io/anaconda3/Anaconda3-2.0.0-Linux-x86.sh）或者 64-bit version（http://bit.ly/a/warning?url=http%3a%2f%2frepo%2econtinuum%2eio%2fanaconda3%2fAnaconda3%2d2%2e0%2e0%2dWindows%2dx86%5f64%2eexe&hash=win-64b）进行下载。下载结束后，执行它（一个大型 shell 脚本）即可。

> 确保你下载的文件是以 Anaconda3 开头的。如果文件名仅仅是以 Anaconda 开头，那么你下载的应该是 Python 2 对应的版本，它不是我们想要的。

Anaconda 将所有文件都安装到属于它自己的目录中（home 路径下 anaconda 目录中）。这意味着它的安装不会对你电脑上的任何 Python 版本产生影响，同时你也不需要特殊的权限（像 `admin` 和 `root` 之类的）来执行安装。

如果你想要知道 Anaconda 包含哪些 Python 包，可以访问 Anaconda 文档页（http://docs.continuum.io/anaconda/pkg-docs.html），点击文档顶部选择框中的“Python version：3.4”即可。我上次查看时总共列出了 141 个包。

安装完毕 Anaconda 2 后，你可以看看这位“圣诞老人”都在你的电脑里放了些什么小礼物。执行下面的命令：

```
$ ./conda list

# packages in environment at /Users/williamlubanovic/anaconda:
#
anaconda                  2.0.0                np18py34_0
argcomplete               0.6.7                    py34_0
astropy                   0.3.2                np18py34_0
backports.ssl-match-hostname 3.4.0.2                  <pip>
beautiful-soup            4.3.1                    py34_0
beautifulsoup4            4.3.1                   <pip>
binstar                   0.5.3                    py34_0
bitarray                  0.8.1                    py34_0
blaze                     0.5.0                np18py34_0
blz                       0.6.2                np18py34_0
bokeh                     0.4.4                np18py34_1
cdecimal                  2.3                      py34_0
colorama                  0.2.7                    py34_0
conda                     3.5.2                    py34_0
conda-build               1.3.3                    py34_0
configobj                 5.0.5                    py34_0
curl                      7.30.0                        2
cython                    0.20.1                   py34_0
datashape                 0.2.0                np18py34_1
dateutil                  2.1                      py34_2
```

```
docutils                  0.11                       py34_0
dynd-python               0.6.2                  np18py34_0
flask                     0.10.1                     py34_1
freetype                  2.4.10                          1
future                    0.12.1                     py34_0
greenlet                  0.4.2                      py34_0
h5py                      2.3.0                  np18py34_0
hdf5                      1.8.9                           2
ipython                   2.1.0                      py34_0
ipython-notebook          2.1.0                      py34_0
ipython-qtconsole         2.1.0                      py34_0
itsdangerous              0.24                       py34_0
jdcal                     1.0                        py34_0
jinja2                    2.7.2                      py34_0
jpeg                      8d                              1
libdynd                   0.6.2                           0
libpng                    1.5.13                          1
libsodium                 0.4.5                           0
libtiff                   4.0.2                           0
libxml2                   2.9.0                           1
libxslt                   1.1.28                          2
llvm                      3.3                             0
llvmpy                    0.12.4                     py34_0
lxml                      3.3.5                      py34_0
markupsafe                0.18                       py34_0
matplotlib                1.3.1                  np18py34_1
mock                      1.0.1                      py34_0
multipledispatch          0.4.3                      py34_0
networkx                  1.8.1                      py34_0
nose                      1.3.3                      py34_0
numba                     0.13.1                 np18py34_0
numexpr                   2.3.1                  np18py34_0
numpy                     1.8.1                      py34_0
openpyxl                  2.0.2                      py34_0
openssl                   1.0.1g                          0
pandas                    0.13.1                 np18py34_0
patsy                     0.2.1                  np18py34_0
pillow                    2.4.0                      py34_0
pip                       1.5.6                      py34_0
ply                       3.4                        py34_0
psutil                    2.1.1                      py34_0
py                        1.4.20                     py34_0
pycosat                   0.6.1                      py34_0
pycparser                 2.10                       py34_0
pycrypto                  2.6.1                      py34_0
pyflakes                  0.8.1                      py34_0
pygments                  1.6                        py34_0
pyparsing                 2.0.1                      py34_0
pyqt                      4.10.4                     py34_0
pytables                  3.1.1                  np18py34_0
pytest                    2.5.2                      py34_0
python                    3.4.1                           0
python-dateutil           2.1                         <pip>
python.app                1.2                        py34_2
pytz                      2014.3                     py34_0
```

```
pyyaml                    3.11                       py34_0
pyzmq                     14.3.0                     py34_0
qt                        4.8.5                           3
readline                  6.2                             2
redis                     2.6.9                           0
redis-py                  2.9.1                      py34_0
requests                  2.3.0                      py34_0
rope                      0.9.4                      py34_1
rope-py3k                 0.9.4                       <pip>
runipy                    0.1.0                      py34_0
scikit-image              0.9.3                  np18py34_0
scipy                     0.14.0                 np18py34_0
setuptools                3.6                        py34_0
sip                       4.15.5                     py34_0
six                       1.6.1                      py34_0
sphinx                    1.2.2                      py34_0
spyder                    2.3.0rc1                   py34_0
spyder-app                2.3.0rc1                   py34_0
sqlalchemy                0.9.4                      py34_0
sqlite                    3.8.4.1                         0
ssl_match_hostname        3.4.0.2                    py34_0
sympy                     0.7.5                      py34_0
tables                    3.1.1                       <pip>
tk                        8.5.15                          0
tornado                   3.2.1                      py34_0
ujson                     1.33                       py34_0
werkzeug                  0.9.4                      py34_0
xlrd                      0.9.3                      py34_0
xlsxwriter                0.5.5                      py34_0
yaml                      0.1.4                           1
zeromq                    4.0.4                           0
zlib                      1.2.7                           1
```

D.3　安装并使用pip和virtualenv

`pip` 是最受欢迎的管理第三方（非标准库）Python 包的包。这么实用的一款工具竟然不是标准 Python 的一部分实在是让人觉得很烦恼，因为我们不得不自己手动下载安装。我的一个朋友曾经说过：这简直就是残忍恼人的仪式。好消息是从 Python 3.4 开始，`pip` 已经被包含到标准 Python 中了！

`virtualenv` 经常与 `pip` 一起使用，它允许我们将 Python 包安装到指定的路径（文件夹）中以避免它与一些已经安装了的 Python 包互相影响。它的精髓在于，即使你没有权限修改已安装的 Python 包，你仍然可以随心所欲地使用任何你喜欢的 Python 工具[2]。

如果你安装了 Python 3，但发现使用的仍然是 Python 2 版本的 `pip`，下面的命令可以帮助你在 Linux 或 OS X 下获取 Python 3 对应的版本：

```
$ curl -O http://python-distribute.org/distribute_setup.py
$ sudo python3 distribute_setup.py
```

注 2：既然不会互相影响，重新安装一份即可。——译者注

```
$ curl -O https://raw.github.com/pypa/pip/master/contrib/get-pip.py
$ sudo python3 get-pip.py
```

上面的命令会将 `pip-3.3` 安装到你的 Python 3 安装路径下的 `bin` 目录中。之后你就可以使用 `pip-3.3` 而不是 Python 2 的 `pip` 来安装第三方包了。

下面列出一些有关 `pip` 和 `virtualenv` 教程的链接：

- A non-magical introduction to Pip and Virtualenv for Python beginners（http://www.dabapps.com/blog/introduction-to-pip-and-virtualenv-python/）
- The hitchhiker's guide to packaging: pip（http://the-hitchhikers-guide-to-packaging.readthedocs.org/en/latest/pip.html）

D.4　安装并使用conda

目前为止，`pip` 下载得到的大多是源文件而不是编译完成的二进制文件，但有些 Python 模块并不是用 Python 编写的而是用 C 语言库编写的，这意味着我们不得不进行额外的编译工作，非常麻烦。最近，Anaconda 的开发者们编写了 `conda`（http://www.continuum.io/blog/conda），旨在解决使用 `pip` 和其他工具时存在的各种问题。`pip` 是一个 Python 专属的包管理器，但 `conda` 可以运行在任何软件和语言环境中。`conda` 还同时解决了使用 `pip` 时不得不同时使用 `virtualenv` 这样的工具来保证安装的包之间互不影响的问题。

如果你安装了 Anaconda，`conda` 就已经附带安装完毕。如果你没有安装 Anaconda，可以从 miniconda 网页（http://conda.pydata.org/miniconda.html）下载 `conda` 和与它绑定的 Python 3。和 Anaconda 一样，请确保你下载的文件以 `Miniconda3` 开头，仅以 `Miniconda` 开头的是 Python 2 对应的版本。

`conda` 依赖 `pip`。尽管 `conda` 有自己的包仓库（https://binstar.org/），但像 `conda search` 这样的命令在执行时除了搜索 `conda` 包仓库中的内容外，还会搜索 PyPi 库（http://pypi.python.org/）。如果你在使用 `pip` 时遇到了问题，不妨试试 `conda`。

附录 E

习题解答

E.1 第1章“Python初探”

(1) 如果你还没有安装 Python 3，现在就立刻动手。具体方法请阅读附录 D。

(2) 启动 Python 3 交互式解释器。再说一次，具体方法请阅读附录 D。它会打印出几行信息和一行 >>>，这是你输入 Python 命令的提示符。

下面是在我的 MacBook Pro 上显示的内容：

```
$ python
Python 3.3.0 (v3.3.0:bd8afb90ebf2, Sep 29 2012, 01:25:11)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(3) 随便玩玩解释器。可以用它来计算 **8 * 9**。按下回车来查看结果，Python 应该会打印出 72。

```
>>> 8 * 9
72
```

(4) 输入数字 **47** 并按下回车，解释器有没有在下一行打印出 47？

```
>>> 47
47
```

(5) 现在输入 **print(47)** 并按下回车，解释器有没有在下一行打印出 47？

```
>>> print(47)
47
```

E.2　第2章“Python基本元素：数字、字符串和变量”

(1) 一个小时有多少秒？这里，请把交互式解释器当作计算器使用，将每分钟的秒数（60）乘以每小时的分钟数（60）得到结果。

```
>>> 60 * 60
3600
```

(2) 将上一个练习得到的结果（每小时的秒数）赋值给名为 seconds_per_hour 的变量。

```
>>> seconds_per_hour = 60 * 60
>>> seconds_per_hour
3600
```

(3) 一天有多少秒？用你的 seconds_per_hour 变量进行计算。

```
>>> seconds_per_hour * 24
86400
```

(4) 再次计算每天的秒数，但这一次将结果存储在名为 seconds_per_day 的变量中。

```
>>> seconds_per_day = seconds_per_hour * 24
>>> seconds_per_day
86400
```

(5) 用 seconds_per_day 除以 seconds_per_hour，使用浮点除法（/）。

```
>>> seconds_per_day / seconds_per_hour
24.0
```

(6) 用 seconds_per_day 除以 seconds_per_hour，使用整数除法（//）。除了末尾的 .0，本练习所得结果是否与前一个练习用浮点数除法得到的结果一致？

```
>>> seconds_per_day // seconds_per_hour
24
```

E.3

(1) 创建一个叫作 years_list 的列表，存储从你出生的那一年到你五岁那一年的年份。例如，如果你是 1980 年出生的，那么你的列表应该是 years_list = [1980, 1981, 1982, 1983, 1984, 1985]。

假设你出生在 1980 年，输入如下所示：

```
>>> years_list = [1980, 1981, 1982, 1983, 1984, 1985]
```

(2) 在 years_list 中，哪一年是你三岁生日那年？别忘了，你出生的第一年算 0 岁。

你需要的偏移量为 3，如果你出生在 1980 年，那么：

```
>>> years_list[3]
1983
```

(3) 在 `years_list` 中，哪一年你的年纪最大？

你需要得到列表中的最后一项，因此使用偏移量 `-1`，或者你也可以使用偏移量 `5`，因为你已经提前知道列表中有 6 项，但是 `-1` 返回任何大小列表的最后一项。对于一个 1980 年代的人：

```
>>> years_list[-1]
1985
```

(4) 创建一个名为 `things` 的列表，包含以下三个元素：`"mozzarella"`、`"cinderella"` 和 `"salmonella"`。

```
>>> things = ["mozzarella", "cinderella", "salmonella"]
>>> things
['mozzarella', 'cinderella', 'salmonella']
```

(5) 将 `things` 中代表人名的字符串变成首字母大写形式，并打印整个列表。看看列表中的元素改变了么？

下面的方法实现了单词首字母大写，但是没有在列表中改变它：

```
>>> things[1].capitalize()
'Cinderella'
>>> things
['mozzarella', 'cinderella', 'salmonella']
```

如果你想在列表中改变它，应该将它重新赋值回列表：

```
>>> things[1] = things[1].capitalize()
>>> things
['mozzarella', 'Cinderella', 'salmonella']
```

(6) 将 `things` 中代表奶酪的元素全部改成大写，并打印整个列表。

```
>>> things[0] = things[0].upper()
>>> things
['MOZZARELLA', 'Cinderella', 'salmonella']
```

(7) 将代表疾病的元素从 `things` 中删除，收好你因此得到的诺贝尔奖，并打印列表。

按照值删掉它：

```
>>> things.remove("salmonella")
>>> things
['MOZZARELLA', 'Cinderella']
```

因为它是列表中的最后一项，所以下面的方法也是可行的：

```
>>> del things[-1]
```

或者从列表开始处按照偏移量删掉它：

```
>>> del things[2]
```

(8) 创建一个名为 `surprise` 的列表，包含以下三个元素：`"Groucho"`、`"Chico"` 和 `"Harpo"`。

```
>>> surprise = ['Groucho', 'Chico', 'Harpo']
>>> surprise
['Groucho', 'Chico', 'Harpo']
```

(9) 将 surprise 列表的最后一个元素变成小写，翻转过来，再将首字母变成大写。

```
>>> surprise[-1] = surprise[-1].lower()
>>> surprise[-1] = surprise[-1][::-1]
>>> surprise[-1].capitalize()
'Oprah'
```

(10) 创建一个名为 e2f 的英法字典并打印出来。这里提供一些单词对：dog 是 chien、cat 是 chat 以及 walrus 是 morse。

```
>>> e2f = {'dog': 'chien', 'cat': 'chat', 'walrus': 'morse'}
>>> e2f
{'cat': 'chat', 'walrus': 'morse', 'dog': 'chien'}
```

(11) 使用你的仅包含三个词的字典 e2f 查询并打印出 walrus 对应的法语词。

```
>>> e2f['walrus']
'morse'
```

(12) 利用 e2f 创建一个名为 f2e 的法英字典。注意要使用 items 方法。

```
>>> f2e = {}
>>> for english, french in e2f.items():
    f2e[french] = english
>>> f2e
{'morse': 'walrus', 'chien': 'dog', 'chat': 'cat'}
```

(13) 使用 f2e，查询并打印法语词 chien 对应的英文词。

```
>>> f2e['chien']
'dog'
```

(14) 创建并打印由 e2f 的键组成的英语单词集合。

```
>>> set(e2f.keys())
{'cat', 'walrus', 'dog'}
```

(15) 建立一个名为 life 的多级字典，将下面这些字符串作为顶级键：'animals'、'plants' 以及 'others'。令 'animals' 键指向另一个字典，这个字典包含键 'cats'、'octopi' 以及 'emus'。令 'cats' 键指向一个字符串列表，这个列表包括 'Henri'、'Grumpy' 和 'Lucy'。让其余的键都指向空字典。

这是一道比较难的题，如果第一眼看到，不要觉得不舒服：

```
>>> life = {
...     'animals': {
...         'cats': [
...             'Henri', 'Grumpy', 'Lucy'
...             ],
...         'octopi': {},
...         'emus': {}
```

```
...         },
...     'plants': {},
...     'other': {}
...     }
>>>
```

(16) 打印 life 的顶级键。

```
>> print(life.keys())
dict_keys(['animals', 'other', 'plants'])
```

Python 3 包含 dict_keys 的项，把它作为普通的列表打印输出：

```
>>> print(list(life.keys()))
['animals', 'other', 'plants']
```

顺便提一句，在代码中多加空格可以提高可读性：

```
>>> print ( list ( life.keys() ) )
['animals', 'other', 'plants']
```

(17) 打印 life['animals'] 的全部键。

```
>>> print(life['animals'].keys())
dict_keys(['cats', 'octopi', 'emus'])
```

(18) 打印 life['animals']['cats'] 的值。

```
>>> print(life['animals']['cats'])
['Henri', 'Grumpy', 'Lucy']
```

E.4 第4章“Python外壳：代码结构”

(1) 将 7 赋值给变量 guess_me，然后写一段条件判断（if、else 和 elif）的代码：如果 guess_me 小于 7 输出 'too low'，大于 7 则输出 'too high'，等于 7 则输出 'just right'。

```
guess_me = 7
if guess_me < 7:
    print('too low')
elif guess_me > 7:
    print('too high')
else:
    print('just right')
```

执行这段代码得到如下结果：

```
just right
```

(2) 将 7 赋值给变量 guess_me，再将 1 赋值给变量 start。写一段 while 循环代码比较 start 和 guess_me：如果 start 小于 guess_me 则输出 'too low'，如果等于则输出 'found it'，如果大于则输出 'oops'，然后终止循环。在每次循环结束时自增 start。

```
guess_me = 7
start = 1
while True:
    if start < guess_me:
        print('too low')
    elif start == guess_me:
        print('found it!')
        break
    elif start > guess_me:
        print('oops')
        break
    start += 1
```

如果代码正确执行，结果如下所示：

```
too low
too low
too low
too low
too low
too low
found it!
```

注意 `elif start > guess_me:` 这一行可以只用简单的 `else:`，因为 `start` 不小于等于 `guess_me` 即大于，至少在这是对的。

(3) 使用 `for` 循环输出列表 `[3, 2, 1, 0]` 的值。

```
>>> for value in [3, 2, 1, 0]:
...     print(value)
...
3
2
1
0
```

(4) 使用列表推导生成 10 以内（`range(10)`）偶数的列表。

```
>>> even = [number for number in range(10) if number % 2 == 0]
>>> even
[0, 2, 4, 6, 8]
```

(5) 使用字典推导创建字典 `squares`。把 0~9 内的整数作为键，每个键的平方作为对应的值。

```
>>> squares = {key: key*key for key in range(10)}
>>> squares
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16, 5: 25, 6: 36, 7: 49, 8: 64, 9: 81}
```

(6) 使用集合推导创建集合 `odd`，包含 0~9 内（`range(10)`）的奇数。

```
>>> odd = {number for number in range(10) if number % 2 == 1}
>>> odd
{1, 3, 9, 5, 7}
```

(7) 使用生成器推导返回字符串 `'Got '` 和 0~9 内的一个整数，使用 `for` 循环进行迭代。

```
>> for thing in ('Got %s' % number for number in range(10)):
...     print(thing)
...
Got 0
Got 1
Got 2
Got 3
Got 4
Got 5
Got 6
Got 7
Got 8
Got 9
```

(8) 定义函数 `good()`：返回列表 `['Harry', 'Ron', 'Hermione']`。

```
>>> def good():
...     return ['Harry', 'Ron', 'Hermione']
...
>>> good()
['Harry', 'Ron', 'Hermione']
```

(9) 定义一个生成器函数 `get_odds()`：返回 0~9 内的奇数。使用 `for` 循环查找并输出返回的第三个值。

```
>>> def get_odds():
...     for number in range(1, 10, 2):
...         yield number
...
>>> for count, number in enumerate(get_odds(), 1):
...     if count == 3:
...         print("The third odd number is", number)
...         break
...
The third odd number is 5
```

(10) 定义一个装饰器 `test`：当一个函数被调用时输出 `'start'`，当函数结束时输出 `'end'`。

```
>>> def test(func):
...     def new_func(*args, **kwargs):
...         print('start')
...         result = func(*args, **kwargs)
...         print('end')
...         return result
...     return new_func
...
>>>
>>> @test
... def greeting():
...     print("Greetings, Earthling")
...
>>> greeting()
start
```

```
Greetings, Earthling
end
```

(11) 定义一个异常 OopsException：编写代码捕捉该异常，并输出 'Caught an oops'。

```
>>> class OopsException(Exception):
...     pass
...
>>> raise OopsException()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
__main__.OopsException
>>>
>>> try:
...     raise OopsException
... except OopsException:
...     print('Caught an oops')
...
Caught an oops
```

(12) 使用函数 zip() 创建字典 movies：匹配两个列表 titles = ['Creature of Habit', 'Crewel Fate'] 和 plots = ['A nun turns into a monster', 'A haunted yarn shop']。

```
>>> titles = ['Creature of Habit', 'Crewel Fate']
>>> plots = ['A nun turns into a monster', 'A haunted yarn shop']
>>> movies = dict(zip(titles, plots))
>>> movies
{'Crewel Fate': 'A haunted yarn shop', 'Creature of Habit': 'A nun turns
into a monster'}
```

E.5 第5章“Python盒子：模块、包和程序”

(1) 创建文件 zoo.py。在文件中定义函数 hours：输出字符串 'Open 9-5 daily'。然后使用交互式解释器导入模块 zoo，调用函数 hours。下面是文件 zoo.py：

```
def hours():
    print('Open 9-5 daily')
```

现在，在解释器中导入它：

```
>>> import zoo
>>> zoo.hours()
Open 9-5 daily
```

(2) 在交互式解释器，把模块 zoo 作为 menagerie 导入，然后调用函数 hours()。

```
>>> import zoo as menagerie
>>> menagerie.hours()
Open 9-5 daily
```

(3) 继续在解释器中，直接从模块 zoo 导入函数 hours()，然后调用。

```
>>> from zoo import hours
>>> hours()
Open 9-5 daily
```

(4) 把函数 hours() 作为 info 导入，然后调用它。

```
>>> from zoo import hours as info
>>> info()
Open 9-5 daily
```

(5) 创建字典 plain：包含键值对 'a':1、'b':2 和 'c':3，然后输出它。

```
>>> plain = {'a': 1, 'b': 2, 'c': 3}
>>> plain
{'a': 1, 'c': 3, 'b': 2}
```

(6) 创建有序字典 fancy：键值对和 (5) 相同，然后输出它。输出顺序和 plain 相同吗？

```
>>> from collections import OrderedDict
>>> fancy = OrderedDict([('a', 1), ('b', 2), ('c', 3)])
>>> fancy
OrderedDict([('a', 1), ('b', 2), ('c', 3)])
```

(7) 创建默认字典 dict_of_lists，传入参数 list：给 dict_of_lists['a'] 赋值 'something for a'，输出 dict_of_lists['a'] 的值。

```
>>> from collections import defaultdict
>>> dict_of_lists = defaultdict(list)
>>> dict_of_lists['a'].append('something for a')
>>> dict_of_lists['a']
['something for a']
```

E.6 第6章“对象和类”

(1) 创建一个名为 Thing 的空类并将它打印出来。接着创建一个属于该类的对象 example，同样将它打印出来。看看这两次打印结果是一样的还是不同的？

```
>>> class Thing:
...     pass
...
>>> print(Thing)
<class '__main__.Thing'>
>>> example = Thing()
>>> print(example)
<__main__.Thing object at 0x1006f3fd0>
```

(2) 创建一个新类 Thing2，将 'abc' 赋值给类特性 letters，打印 letters。

```
>>> class Thing2:
...     letters = 'abc'
...
>>> print(Thing2.letters)
abc
```

(3) 再创建一个新类 Thing3。这次将 'xyz' 赋值给实例（对象）特性 letters，并打印 letters。看看你是不是必须先创建一个对象才可以进行打印操作？

```
>>> class Thing3:
...     def __init__(self):
...         self.letters = 'xyz'
...
```

变量 letters 属于类 Thing3 的任何对象，而不是 Thing3 类本身：

```
>>> print(Thing3.letters)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: type object 'Thing3' has no attribute 'letters'
>>> something = Thing3()
>>> print(something.letters)
xyz
```

(4) 创建一个名为 Element 的类，它包含实例属性 name、symbol 和 number。使用 'Hydrogen'、'H' 和 1 实例化一个对象 hydrogen。

```
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.name = name
...         self.symbol = symbol
...         self.number = number
...
>>> hydrogen = Element('Hydrogen', 'H', 1)
```

(5) 创建一个字典，包含这些键值对：'name': 'Hydrogen'、'symbol': 'H' 和 'number': 1。然后用这个字典实例化 Element 类的对象 hydrogen。

首先创建该字典：

```
>>> el_dict = {'name': 'Hydrogen', 'symbol': 'H', 'number': 1}
```

虽然会编写较多代码，但这是可行的：

```
>>> hydrogen = Element(el_dict['name'], el_dict['symbol'], el_dict['number'])
```

检查一下实例化的结果：

```
>>> hydrogen.name
'Hydrogen'
```

然而，你可以直接从字典初始化对象，因为它的键名称是和 __init__ 参数相匹配的（参考第 3 章关于关键字参数的讨论）：

```
>>> hydrogen = Element(**el_dict)
>>> hydrogen.name
'Hydrogen'
```

(6) 为 Element 类定义个 dump() 方法，用于打印对象的属性（name、symbol 和 number）。使用这个新类创建一个对象 hydrogen 并用 dump() 打印。

```
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.name = name
```

```
...         self.symbol = symbol
...         self.number = number
...     def dump(self):
...         print('name=%s, symbol=%s, number=%s' %
...             (self.name, self.symbol, self.number))
...
>>> hydrogen = Element(**el_dict)
>>> hydrogen.dump()
name=Hydrogen, symbol=H, number=1
```

(7) 调用 `print(hydrogen)`，然后修改 `Element` 的定义，将 `dump` 方法的名字改为 `__str__`。再次创建一个 `hydrogen` 对象并调用 `print(hydrogen)`，观察输出结果。

```
>>> print(hydrogen)
<__main__.Element object at 0x1006f5310>
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.name = name
...         self.symbol = symbol
...         self.number = number
...     def __str__(self):
...         return ('name=%s, symbol=%s, number=%s' %
...             (self.name, self.symbol, self.number))
...
>>> hydrogen = Element(**el_dict)
>>> print(hydrogen)
name=Hydrogen, symbol=H, number=1
```

`__str__()` 是 Python 的一个魔术方法，`print` 函数调用一个对象的 `__str__()` 方法获取它的字符串表示。如果类中没有定义 `__str__()` 方法，它会采用父类的默认方法，返回类似于 `<__main__.Element object at 0x1006f5310>` 的一个字符串。

(8) 修改 `Element` 使得 `name`、`symbol` 和 `number` 特性都变成私有的。为它们各定义一个 getter 属性（property）来返回各自的值。

```
>>> class Element:
...     def __init__(self, name, symbol, number):
...         self.__name = name
...         self.__symbol = symbol
...         self.__number = number
...     @property
...     def name(self):
...         return self.__name
...     @property
...     def symbol(self):
...         return self.__symbol
...     @property
...     def number(self):
...         return self.__number
...
>>> hydrogen = Element('Hydrogen', 'H', 1)
>>> hydrogen.name
'Hydrogen'
>>> hydrogen.symbol
```

```
'H'
>>> hydrogen.number
1
```

(9) 定义三个类：Bear、Rabbit 和 Octothorpe。对每个类都只定义一个方法 eats()，分别返回 'berries'(Bear)、'clover'(Rabbit) 和 'campers'(Octothorpo)。为每个类创建一个对象并输出它们各自吃的食物（调用 eats()）。

```
>> class Bear:
...     def eats(self):
...         return 'berries'
...
>>> class Rabbit:
...     def eats(self):
...         return 'clover'
...
>>> class Octothorpe:
...     def eats(self):
...         return 'campers'
...
>>> b = Bear()
>>> r = Rabbit()
>>> o = Octothorpe()
>>> print(b.eats())
berries
>>> print(r.eats())
clover
>>> print(o.eats())
campers
```

(10) 定义三个类：Laser、Claw 以及 SmartPhone。每个类都仅有一个方法 does()，分别返回 'disintegrate'(Laser)、'cursh'(Claw) 以及 'ring'(SmartPhone)。接着定义 Robot 类，包含上述三个类的实例（对象）各一个。给 Robot 定义 does() 方法，用于输出它各部分的功能。

```
>>> class Laser:
...     def does(self):
...         return 'disintegrate'
...
>>> class Claw:
...     def does(self):
...         return 'crush'
...
>>> class SmartPhone:
...     def does(self):
...         return 'ring'
...
>>> class Robot:
...     def __init__(self):
...         self.laser = Laser()
...         self.claw = Claw()
...         self.smartphone = SmartPhone()
...     def does(self):
...         return '''I have many attachments:
```

```
... My laser, to %s.
... My claw, to %s.
... My smartphone, to %s.''' % (
...     self.laser.does(),
...     self.claw.does(),
...     self.smartphone.does() )
...
>>> robbie = Robot()
>>> print( robbie.does() )
I have many attachments:
My laser, to disintegrate.
My claw, to crush.
My smartphone, to ring.
```

E.7 第7章“像高手一样玩转数据”

(1) 创建一个 Unicode 字符串 `mystery` 并将它的值设为 `'\U0001f4a9'`。打印 `mystery`，并查看 `mystery` 的 Unicode 名称。

```
>>> import unicodedata
>>> mystery = '\U0001f4a9'
>>> mystery
'□'
>>> unicodedata.name(mystery)
'PILE OF POO'
```

它们还会变成什么呢？

(2) 使用 UTF-8 对 `mystery` 进行编码，存入字节型变量 `pop_bytes`，并将它打印出来。

```
>>> pop_bytes = mystery.encode('utf-8')
>>> pop_bytes
b'\xf0\x9f\x92\xa9'
```

(3) 使用 UTF-8 对 `pop_bytes` 进行解码，存入字符串型变量 `pop_string`，并将它打印出来，看看它与 `mystery` 是否一致？

```
>>> pop_string = pop_bytes.decode('utf-8')
>>> pop_string
'□'
>>> pop_string == mystery
True
```

(4) 使用旧式格式化方法生成下面的诗句，把 `'roast beef'`、`'ham'`、`'head'` 以及 `'clam'` 依次插入字符串：

```
My kitty cat likes %s,
My kitty cat likes %s,
My kitty cat fell on his %s
And now thinks he's a %s.

>>> poem = '''
```

```
... My kitty cat likes %s,
... My kitty cat likes %s,
... My kitty cat fell on his %s
... And now thinks he's a %s.
... '''
>>> args = ('roast beef', 'ham', 'head', 'clam')
>>> print(poem % args)

My kitty cat likes roast beef,
My kitty cat likes ham,
My kitty cat fell on his head
And now thinks he's a clam.
```

(5) 使用新式格式化方法生成下面的套用信函，将下面的字符串存储为 letter（后面的练习中会用到）：

```
Dear {salutation} {name},

Thank you for your letter. We are sorry that our {product} {verbed} in your
{room}. Please note that it should never be used in a {room}, especially
near any {animals}.

Send us your receipt and {amount} for shipping and handling. We will send
you another {product} that, in our tests, is {percent}% less likely to
have {verbed}.

Thank you for your support.

Sincerely,
{spokesman}
{job_title}

>>> letter = '''
... Dear {salutation} {name},
...
... Thank you for your letter. We are sorry that our {product} {verb} in your
... {room}. Please note that it should never be used in a {room}, especially
... near any {animals}.
...
... Send us your receipt and {amount} for shipping and handling. We will send
... you another {product} that, in our tests, is {percent}% less likely to
... have {verbed}.
...
... Thank you for your support.
...
... Sincerely,
... {spokesman}
... {job_title}
... '''
```

(6) 创建一个字典 response，包含以下键：'salutaion'、'name'、'product'、verved（动词过去式）、'room'、'animals'、'percent'、'spokesman' 以及 'job_title'。设定这些键对应的值，并打印由 response 的值填充的 letter。

```
>>> response = {
...     'salutation': 'Colonel',
...     'name': 'Hackenbush',
...     'product': 'duck blind',
...     'verbed': 'imploded',
...     'room': 'conservatory',
...     'animals': 'emus',
...     'amount': '$1.38',
...     'percent': '1',
...     'spokesman': 'Edgar Schmeltz',
...     'job_title': 'Licensed Podiatrist'
...     }
...
>>> print( letter.format(**response) )

Dear Colonel Hackenbush,

Thank you for your letter. We are sorry that our duck blind imploded in your
conservatory. Please note that it should never be used in a conservatory,
especially near any emus.

Send us your receipt and $1.38 for shipping and handling. We will send
you another duck blind that, in our tests, is 1% less likely to have imploded.

Thank you for your support.

Sincerely,
Edgar Schmeltz
Licensed Podiatrist
```

(7) 正则表达式在处理文本上非常方便。在这个练习中，我们会对示例文本尝试做各种各样的操作。我们示例文本是一首名为 *Ode on the Mammoth Cheese* 的诗，它的作者是 James McIntyre，写于 1866 年，出于对当时安大略湖手工制造的 7000 磅的巨型奶酪的敬意，它当时甚至在全球巡回展出。如果你不愿意自己一字一句敲出来，直接百度一下粘贴到你的 Python 代码里即可。你也可以从 Project Gutenberg（http://www.gutenberg.org/ebooks/36068?msg=welcome_stranger）找到。我们将这个字符串命名为 `mammoth`。

```
>>> mammoth = '''
... We have seen thee, queen of cheese,
... Lying quietly at your ease,
... Gently fanned by evening breeze,
... Thy fair form no flies dare seize.
...
... All gaily dressed soon you'll go
... To the great Provincial show,
... To be admired by many a beau
... In the city of Toronto.
...
... Cows numerous as a swarm of bees,
... Or as the leaves upon the trees,
... It did require to make thee please,
... And stand unrivalled, queen of cheese.
```

```
...
... May you not receive a scar as
... We have heard that Mr. Harris
... Intends to send you off as far as
... The great world's show at Paris.
...
... Of the youth beware of these,
... For some of them might rudely squeeze
... And bite your cheek, then songs or glees
... We could not sing, oh! queen of cheese.
...
... We'rt thou suspended from balloon,
... You'd cast a shade even at noon,
... Folks would think it was the moon
... About to fall and crush them soon.
... '''
```

(8) 引入 re 模块以便使用正则表达式相关函数。使用 re.findall() 打印出所有以 'c' 开头的单词。

首先对所要匹配的模式定义变量 pat，然后在 mammoth 中查找：

```
>>> import re
>>> re = r'\bc\w*'
>>> re.findall(pat, mammoth)
['cheese', 'city', 'cheese', 'cheek', 'could', 'cheese', 'cast', 'crush']
```

\b 代表以单词之间的分隔符作为开始，使用它一般用于指定单词的开始或者结束，字母 c 是我们要查找单词的首字母。\w 代表任意单词字符（包括字母、数字和下划线）。* 表示一个或者多个字符。综合起来，它用来查找以字母 c 开头的单词，包括 'c' 本身。如果你不使用原始字符串（在第一个引号前加 r），Python 会把 \b 解释为退格字符，查找会神奇地挂掉：

```
>>> pat = '\bc\w*'
>>> re.findall(pat, mammoth)
[]
```

(9) 找到所有以 c 开头的 4- 字母单词。

```
>>> pat = r'\bc\w{3}\b'
>>> re.findall(pat, mammoth)
['city', 'cast']
```

你需要最后的 \b 来指明单词的结束。否则，你会得到所有以 c 开头并且至少有四个字母的单词的前四个字母：

```
>>> pat = r'\bc\w{3}'
>>> re.findall(pat, mammoth)
['chee', 'city', 'chee', 'chee', 'coul', 'chee', 'cast', 'crus']
```

(10) 找到所有以 r 结尾的单词。

下面代码使用要小心，对于以 r 结尾的单词会得到完美的结果：

```
>>> pat = r'\b\w*r\b'
>>> re.findall(pat,mammoth)
['your', 'fair', 'Or', 'scar', 'Mr', 'far', 'For', 'your', 'or']
```

然而，用在以 l 结尾的单词结果就不好：

```
>>> pat = r'\b\w*l\b'
>>> re.findall(pat,mammoth)
['All', 'll', 'Provincial', 'fall']
```

但是，ll 为什么出现在那儿？ \w 仅仅匹配到字母、数字和下划线，不会匹配到 ASCII 中的撇号 (')。所以，它会从 you'll 抽取到最后的 ll。解决该问题可以把撇号加到要匹配的字符集合。第一次这样做失败了：

```
>>> >>> pat = r'\b[\w']*l\b'
  File "<stdin>", line 1
    pat = r'\b[\w']*l\b'
```

Python 指到了错误附近的位置，但仍然需要花费一段时间发现模式串被两个引号（撇号）同时包括，一个解决方法是加转移字符：

```
>>> pat = r'\b[\w\']*l\b'
>>> re.findall(pat, mammoth)
['All', "you'll", 'Provincial', 'fall']
```

另一种方法是给模式串加双引号：

```
>>> pat = r"\b[\w']*l\b"
>>> re.findall(pat, mammoth)
['All', "you'll", 'Provincial', 'fall']
```

(11) 找到所有包含且仅包含连续 3 个元音的单词。

开始匹配时是一个单词边界符，然后任意数目的字母、三个连续的元音，接下来是任意数目的非元音字符直到单词结束：

```
>>> pat = r'\b\w*[aeiou]{3}[^aeiou]\w*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'beau\nIn', 'queen', 'squeeze', 'queen']
```

上面的匹配看起来是对的，除了字符串 'beau\nIn'。把 mammoth 作为多行的字符串进行搜索，[^aeiou] 匹配任何非元音字符包括换行符。所以要把一些间隔字符加入到忽略集合，例如 \n（\s 匹配到间隔字符）：

```
>>> pat = r'\b\w*[aeiou]{3}[^aeiou\s]\w*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'queen', 'squeeze', 'queen']
```

但这一次没有搜索到 beau，所以还要对模式串进行变动，匹配到三个连续元音之后还要匹配任意数目的非元音。之前的模式串只匹配了一个非元音：

```
>>> pat = r'\b\w*[aeiou]{3}[^aeiou\s]*\w*\b'
>>> re.findall(pat, mammoth)
['queen', 'quietly', 'beau', 'queen', 'squeeze', 'queen']
```

上面所有的内容表明了什么？其中一点是：正则表达式可以完成很多事情，但正确使用它还要多加小心。

(12) 使用 unhexlify() 将下面的十六进制串（出于排版原因将它们拆成两行字符串）转换为 bytes 型变量，命名为 gif：

```
'47494638396101000100800000000000ffffff21f9' +
'0401000000002c000000000100010000020144003b'

>>> import binascii
>>> hex_str = '47494638396101000100800000000000ffffff21f9' + \
...     '0401000000002c000000000100010000020144003b'
>>> gif = binascii.unhexlify(hex_str)
>>> len(gif)
42
```

(13) gif 定义了一个 1 像素的透明 GIF 文件（最常见的图片格式之一）。合法的 GIF 文件开头由 GIF89a 组成，检测一下上面的 gif 是否为合法的 GIF 文件？

```
>>> gif[:6] == b'GIF89a'
True
```

注意，我们使用 b 来定义一个字节串而不是 Unicode 字符串，你可以在字节之间做比较，但是不能用字符串和字节比较：

```
>>> gif[:6] == 'GIF89a'
False
>>> type(gif)
<class 'bytes'>
>>> type('GIF89a')
<class 'str'>
>>> type(b'GIF89a')
<class 'bytes'>
```

(14) GIF 文件的像素宽度是一个 16- 比特的以大端方案存储的整数，偏移量为 6 字节，高度数据的大小与之相同，偏移量为 8。从 gif 中抽取这些信息并打印出来，看看它们是否与预期的一样都为 1？

```
>>> import struct
>>> width, height = struct.unpack('<HH', gif[6:10])
>>> width, height
(1, 1)
```

E.8 第8章“数据的归宿”

(1) 将字符串 'This is a test of the emergency text system' 赋给变量 test1，然后把它写到文件 test.txt。

```
>>> test1 = 'This is a test of the emergency text system'
>>> len(test1)
43
```

下面是如何使用 open、write 和 close 函数实现题目要求：

```
>>> outfile = open('test.txt', 'wt')
>>> outfile.write(test1)
43
>>> outfile.close()
```

或者直接使用 with，避免调用 close（Python 帮你实现）：

```
>>> with open('test.txt', 'wt') as outfile:
...     outfile.write(test1)
...
43
```

(2) 打开文件 test.txt，读文件内容到字符串 test2。test1 和 test2 是一样的吗？

```
>>> with open('test.txt', 'rt') as infile:
...     test2 = infile.read()
...
>>> len(test2)
43
>>> test1 == test2
True
```

(3) 保存这些文本到 test.csv 文件。注意，字段间是通过逗号隔开的，如果字段中含有逗号需要在整个字段加引号。

```
author,book
J R R Tolkien,The Hobbit
Lynne Truss,"Eats, Shoots & Leaves"

>>> text = '''author,book
... J R R Tolkien,The Hobbit
... Lynne Truss,"Eats, Shoots & Leaves"
... '''
>>> with open('test.csv', 'wt') as outfile:
...     outfile.write(text)
...
73
```

(4) 使用 csv 模块和它的 DictReader() 方法读取文件 test.csv 到变量 books。输出变量 books 的值。DictReader() 可以处理第二本书题目中的引号和逗号吗？

```
>>> with open('test.csv', 'rt') as infile:
...     books = csv.DictReader(infile)
...     for book in books:
...         print(book)
...
{'book': 'The Hobbit', 'author': 'J R R Tolkien'}
{'book': 'Eats, Shoots & Leaves', 'author': 'Lynne Truss'}
```

(5) 创建包含下面这些行的 CSV 文件 books.csv：

```
title,author,year
The Weirdstone of Brisingamen,Alan Garner,1960
Perdido Street Station,China Miéville,2000
Thud!,Terry Pratchett,2005
```

```
The Spellman Files,Lisa Lutz,2007
Small Gods,Terry Pratchett,1992

>>> text = '''title,author,year
... The Weirdstone of Brisingamen,Alan Garner,1960
... Perdido Street Station,China Miéville,2000
... Thud!,Terry Pratchett,2005
... The Spellman Files,Lisa Lutz,2007
... Small Gods,Terry Pratchett,1992
... '''
>>> with open('books.csv', 'wt') as outfile:
...     outfile.write(text)
...
201
```

(6) 使用 sqlite3 模块创建一个 SQLite 数据库 books.db 以及包含字段 title（text）、author（text）以及 year（integer）的表单 books。

```
>>> import sqlite3
>>> db = sqlite3.connect('books.db')
>>> curs = db.cursor()
>>> curs.execute('''create table book (title text, author text, year int)''')
<sqlite3.Cursor object at 0x1006e3b90>
>>> db.commit()
```

(7) 读取文件 books.csv，把数据插入到表单 book。

```
>>> import csv
>>> import sqlite3
>>> ins_str = 'insert into book values(?, ?, ?)'
>>> with open('books.csv', 'rt') as infile:
...     books = csv.DictReader(infile)
...     for book in books:
...         curs.execute(ins_str, (book['title'], book['author'], book['year']))
...
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
<sqlite3.Cursor object at 0x1007b21f0>
>>> db.commit()
```

(8) 选择表单 book 中的 title 列，并按照字母表顺序输出。

```
>>> sql = 'select title from book order by title asc'
>>> for row in db.execute(sql):
...     print(row)
...
('Perdido Street Station',)
('Small Gods',)
('The Spellman Files',)
('The Weirdstone of Brisingamen',)
('Thud!',)
```

如果你只想输出 title 的值，不包含引号和逗号，试下这个方法：

```
>>> for row in db.execute(sql):
...     print(row[0])
...
Perdido Street Station
Small Gods
The Spellman Files
The Weirdstone of Brisingamen
Thud!
```

如果你想排序时忽略掉题目开头的 'The'，还需要加额外的 SQL 魔法（语句）：

```
>>> sql = '''select title from book order by
... case when (title like "The %") then substr(title, 5) else title end'''
>>> for row in db.execute(sql):
...     print(row[0])
...
Perdido Street Station
Small Gods
The Spellman Files
Thud!
The Weirdstone of Brisingamen
```

(9) 选择表单 book 中所有的列，并按照出版顺序输出。

```
>>> for row in db.execute('select * from book order by year'):
...     print(row)
...
('The Weirdstone of Brisingamen', 'Alan Garner', 1960)
('Small Gods', 'Terry Pratchett', 1992)
('Perdido Street Station', 'China Miéville', 2000)
('Thud!', 'Terry Pratchett', 2005)
('The Spellman Files', 'Lisa Lutz', 2007)
```

为了打印输出表单 book 的每一行所有的字段，用逗号和空格把它们隔开：

```
>>> for row in db.execute('select * from book order by year'):
...     print(*row, sep=', ')
...
The Weirdstone of Brisingamen, Alan Garner, 1960
Small Gods, Terry Pratchett, 1992
Perdido Street Station, China Miéville, 2000
Thud!, Terry Pratchett, 2005
The Spellman Files, Lisa Lutz, 2007
```

(10) 使用 sqlalchemy 模块连接到 sqlite3 数据库 books.db，按照 (8) 一样，选择表单 book 中的 title 列，并按照字母表顺序输出。

```
>>> import sqlalchemy
>>> conn = sqlalchemy.create_engine('sqlite:///books.db')
>>> sql = 'select title from book order by title asc'
>>> rows = conn.execute(sql)
>>> for row in rows:
...     print(row)
...
('Perdido Street Station',)
('Small Gods',)
```

```
('The Spellman Files',)
('The Weirdstone of Brisingamen',)
('Thud!',)
```

(11) 在你的计算机安装 Redis 服务器（参见附录 D）和 Python 的 redis 库（pip install redis）。创建一个 Redis 的哈希表 test，包含字段 count(1) 和 name('Fester Bestertester')，输出 test 的所有字段。

```
>>> import redis
>>> conn = redis.Redis()
>>> conn.delete('test')
1
>>> conn.hmset('test', {'count': 1, 'name': 'Fester Bestertester'})
True
>>> conn.hgetall('test')
{b'name': b'Fester Bestertester', b'count': b'1'}
```

(12) 自增 test 的 count 字段并输出它。

```
>>> conn.hincrby('test', 'count', 3)
4
>>> conn.hget('test', 'count')
b'4'
```

E.9 第9章“剖析Web”

(1) 如果你还没有安装 flask，现在安装它。这样会自动安装 werkzeug、jinja2 和其他包。

(2) 搭建一个网站框架，使用 Flask 的调试 / 代码重载来开发 Web 服务器。使用主机名 localhost 和默认端口 5000 来启动服务器。如果你电脑的 5000 端口已经被占用，使用其他端口。

下面是文件 flask1.py：

```
from flask import Flask

app = Flask(__name__)

app.run(port=5000, debug=True)
```

开启 Web 服务器引擎：

```
$ python flask1.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

(3) 添加一个 home() 函数来处理对于主页的请求，让它返回字符串 It's alive!。

我们该如何调用 flask2.py 呢？

```
from flask import Flask

app = Flask(__name__)
```

```
@app.route('/')
def home():
    return "It's alive!"

app.run(debug=True)
```

开启服务器：

```
$ python flask2.py
 * Running on http://127.0.0.1:5000/
 * Restarting with reloader
```

最后通过浏览器或者命令行 HTTP 程序（例如 `curl`、`wget` 甚至 `telnet`）进入主页：

```
$ curl http://localhost:5000/
It's alive!
```

(4) 创建一个名为 home.html 的 Jinja2 模板文件，内容如下所示：

```
I'm of course referring to {{thing}}, which is {{height}} feet tall and {{color}}.
```

创建名为 templates 的目录，在该目录下创建包含以上内容的文件 home.html。如果你的 Flask 服务器仍在运行中，它会检测到新的内容并自动重启。

(5) 修改 `home()` 函数，让它使用 home.html 模板。给模板传入三个 `GET` 参数：`thing`、`height` 和 `color`。

下面是文件 flask3.py：

```
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route('/')
def home():
    thing = request.values.get('thing')
    height = request.values.get('height')
    color = request.values.get('color')
    return render_template('home.html',
        thing=thing, height=height, color=color)

app.run(debug=True)
```

在 Web 客户端前往地址 http://localhost:5000/?thing=Octothorpe&height=7&color=green，你应该可以看到如下内容：

```
I'm of course referring to Octothorpe, which is 7 feet tall and green.
```

E.10　第10章“系统”

(1) 把当前日期以字符串形式写入文本文件 today.txt。

```
>>> from datetime import date
>>> now = date.today()
>>> now_str = now.isoformat()
>>> with open('today', 'wt') as output:
...     print(now_str, file=output)
>>>
```

除了 print，你可以使用 output.write(now_str) 作为输出。使用 print 会在文件末尾增加一行空行。

(2) 从 today.txt 中读取字符串到 today_string 中。

```
>>> with open('today', 'rt') as input:
...     today_string = input.read()
...
>>> today_string
'2014-02-04\n'
```

(3) 从 today_string 中解析日期。

```
>>> fmt = '%Y-%m-%d\n'
>>> datetime.strptime(today_string, fmt)
datetime.datetime(2014, 2, 4, 0, 0)
```

如果你在文件末尾写入空行，需要在格式字符串（format string）中匹配它。

(4) 列出当前目录下的文件。

如果你的当前目录为 ohmy，包含三个以动物命名的文件，结果可能是这样的：

```
>>> import os
>>> os.listdir('.')
['bears', 'lions', 'tigers']
```

(5) 列出父目录下的文件。

如果父目录包含两个文件和当前的 ohmy 目录，结果可能是这样的：

```
>>> import os
>>> os.listdir('..')
['ohmy', 'paws', 'whiskers']
```

(6) 使用 multiprocessing 创建三个独立的进程，每一个进程在 0 和 1 之间等待随机的时间，输出当前时间，然后终止进程。

保存下面代码到文件 multi_times.py：

```
import multiprocessing

def now(seconds):
    from datetime import datetime
    from time import sleep
    sleep(seconds)
    print('wait', seconds, 'seconds, time is', datetime.utcnow())

if __name__ == '__main__':
```

```
    import random
    for n in range(3):
        seconds = random.random()
        proc = multiprocessing.Process(target=now, args=(seconds,))
        proc.start()
```

```
$ python multi_times.py
wait 0.4670532005508353 seconds, time is 2014-06-03 05:14:22.930541
wait 0.5908421960431798 seconds, time is 2014-06-03 05:14:23.054925
wait 0.8127669040699719 seconds, time is 2014-06-03 05:14:23.275767
```

(7) 创建一个你出生日的日期对象。

假设你出生在 1982 年 8 月 14 日：

```
>>> my_day = date(1982, 8, 14)
>>> my_day
datetime.date(1982, 8, 14)
```

(8) 你的生日是星期几？

```
>>> my_day.weekday()
5
>>> my_day.isoweekday()
6
```

使用函数 `weekday()`，周一返回 0，周日返回 6。而使用函数 `isoweekday()`，周一返回 1，周日返回 7。因此，这一天是周六。

(9) 你出生 10 000 天的日期是什么时候？

```
>>> from datetime import timedelta
>>> party_day = my_day + timedelta(days=10000)
>>> party_day
datetime.date(2009, 12, 30)
```

如果你的生日真的是那天，你可能失去了一个参加聚会的理由（Party 已经过时了）。

E.11　第11章“并发和网络”

(1) 使用原始的 `socket` 来实现一个获取当前时间的服务。当客户端向服务器发送字符串 time 时，服务器会返回当前日期和时间的 ISO 格式字符串。

下面是实现服务器端的一种方式，udp_time_server.py：

```
from datetime import datetime
import socket

address = ('localhost', 6789)
max_size = 4096

print('Starting the server at', datetime.now())
print('Waiting for a client to call.')
```

```
server = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
server.bind(address)
while True:
    data, client_addr = server.recvfrom(max_size)
    if data == b'time':
        now = str(datetime.utcnow())
        data = now.encode('utf-8')
        server.sendto(data, client_addr)
        print('Server sent', data)
server.close()
```

下面是客户端 udp_time_client.py：

```
import socket
from datetime import datetime
from time import sleep

address    = ('localhost', 6789)
max_size   = 4096

print('Starting the client at', datetime.now())
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
    sleep(5)
    client.sendto(b'time', address)
    data, server_addr = client.recvfrom(max_size)
    print('Client read', data)
client.close()
```

在客户端的循环中加入 `sleep(5)` 以避免数据交换过于迅速。在一个窗口开启服务器进程：

```
$ python udp_time_server.py
Starting the server at 2014-06-02 20:28:47.415176
Waiting for a client to call.
```

在另一个窗口执行客户端：

```
$ python udp_time_client.py
Starting the client at 2014-06-02 20:28:51.454805
```

5 秒钟后，你开始得到两者的输出。下面是来自服务器的前三行：

```
Server sent b'2014-06-03 01:28:56.462565'
Server sent b'2014-06-03 01:29:01.463906'
Server sent b'2014-06-03 01:29:06.465802'
```

以下是来自客户端的前三行输出：

```
Client read b'2014-06-03 01:28:56.462565'
Client read b'2014-06-03 01:29:01.463906'
Client read b'2014-06-03 01:29:06.465802'
```

这两个程序都会一直运行，需要人工进行终止。

(2) 使用 ZeroMQ 的 REQ 和 REP 套接字实现同样的功能。

这是服务器端程序 zmq_time_server.py：

```
import zmq
from datetime import datetime

host = '127.0.0.1'
port = 6789
context = zmq.Context()
server = context.socket(zmq.REP)
server.bind("tcp://%s:%s" % (host, port))
print('Server started at', datetime.utcnow())
while True:
    #  等待客户端的下一个请求
    message = server.recv()
    if message == b'time':
        now = datetime.utcnow()
        reply = str(now)
        server.send(bytes(reply, 'utf-8'))
        print('Server sent', reply)
```

以下是客户端程序 zmq_time_client.py：

```
import zmq
from datetime import datetime
from time import sleep

host = '127.0.0.1'
port = 6789
context = zmq.Context()
client = context.socket(zmq.REQ)
client.connect("tcp://%s:%s" % (host, port))
print('Client started at', datetime.utcnow())
while True:
    sleep(5)
    request = b'time'
    client.send(request)
    reply = client.recv()
    print("Client received %s" % reply)
```

对于原始的 socket，你需要首先启动服务器；而使用 ZeroMQ，先启动服务器或者客户端都是可行的。

```
$ python zmq_time_server.py
Server started at 2014-06-03 01:39:36.933532

$ python zmq_time_client.py
Client started at 2014-06-03 01:39:42.538245
```

大约 15 秒后，服务器会返回一些行 :

```
Server sent 2014-06-03 01:39:47.539878
Server sent 2014-06-03 01:39:52.540659
Server sent 2014-06-03 01:39:57.541403
```

可以在客户端看到：

```
Client received b'2014-06-03 01:39:47.539878'
Client received b'2014-06-03 01:39:52.540659'
Client received b'2014-06-03 01:39:57.541403'
```

(3) 使用 XMLRPC 实现同样的功能。

服务器端，xmlrpc_time_server.py：

```
from xmlrpc.server import SimpleXMLRPCServer

def now():
    from datetime import datetime
    data = str(datetime.utcnow())
    print('Server sent', data)
    return data

server = SimpleXMLRPCServer(("localhost", 6789))
server.register_function(now, "now")
server.serve_forever()
```

客户端，xmlrpc_time_client.py：

```
import xmlrpc.client
from time import sleep

proxy = xmlrpc.client.ServerProxy("http://localhost:6789/")
while True:
    sleep(5)
    data = proxy.now()
    print('Client received', data)
```

启动服务器进程：

```
$ python xmlrpc_time_server.py
```

启动客户端进程：

```
$ python xmlrpc_time_client.py
```

大约 15 秒后，这是服务器端输出的前三行：

```
Server sent 2014-06-03 02:14:52.299122
127.0.0.1 - - [02/Jun/2014 21:14:52] "POST / HTTP/1.1" 200 -
Server sent 2014-06-03 02:14:57.304741
127.0.0.1 - - [02/Jun/2014 21:14:57] "POST / HTTP/1.1" 200 -
Server sent 2014-06-03 02:15:02.310377
127.0.0.1 - - [02/Jun/2014 21:15:02] "POST / HTTP/1.1" 200 -
```

下面是客户端输出的前三行：

```
Client received 2014-06-03 02:14:52.299122
Client received 2014-06-03 02:14:57.304741
Client received 2014-06-03 02:15:02.310377
```

(4) 你可能看过那部很老的《我爱露西》（*I Love Lucy*）电视节目。露西和埃塞尔在一个巧克力工厂里工作（这是传统）。她们落在了运输甜点的传送带后面，所以必须用更快的

速度进行处理。写一个程序来模拟这个过程，程序会把不同类型的巧克力添加到一个 Redis 列表中，露西是一个客户端，对列表执行阻塞的弹出操作。她需要 0.5 秒来处理一块巧克力。打印出时间和露西处理的每块巧克力类型以及剩余巧克力的数量。

redis_choc_supply.py 提供初始的工作：

```
import redis
import random
from time import sleep

conn = redis.Redis()
varieties = ['truffle', 'cherry', 'caramel', 'nougat']
conveyor = 'chocolates'
while True:
    seconds = random.random()
    sleep(seconds)
    piece = random.choice(varieties)
    conn.rpush(conveyor, piece)
```

露西的过程更像是 redis_lucy.py：

```
import redis
from datetime import datetime
from time import sleep

conn = redis.Redis()
timeout = 10
conveyor = 'chocolates'
while True:
    sleep(0.5)
    msg = conn.blpop(conveyor, timeout)
    remaining = conn.llen(conveyor)
    if msg:
        piece = msg[1]
        print('Lucy got a', piece, 'at', datetime.utcnow(),
        ', only', remaining, 'left')
```

任意的顺序打开服务器或者客户端进程，因为露西需要半秒钟处理每一个巧克力，而且平均每隔半秒钟会生产一块巧克力，这是一场追赶着的比赛。开始放入传送带上的巧克力越多，露西的工作难度也越大。

```
$ python redis_choc_supply.py&

$ python redis_lucy.py
Lucy got a b'nougat' at 2014-06-03 03:15:08.721169 , only 4 left
Lucy got a b'cherry' at 2014-06-03 03:15:09.222816 , only 3 left
Lucy got a b'truffle' at 2014-06-03 03:15:09.723691 , only 5 left
Lucy got a b'truffle' at 2014-06-03 03:15:10.225008 , only 4 left
Lucy got a b'cherry' at 2014-06-03 03:15:10.727107 , only 4 left
Lucy got a b'cherry' at 2014-06-03 03:15:11.228226 , only 5 left
Lucy got a b'cherry' at 2014-06-03 03:15:11.729735 , only 4 left
Lucy got a b'truffle' at 2014-06-03 03:15:12.230894 , only 6 left
Lucy got a b'caramel' at 2014-06-03 03:15:12.732777 , only 7 left
Lucy got a b'cherry' at 2014-06-03 03:15:13.234785 , only 6 left
```

```
Lucy got a b'cherry' at 2014-06-03 03:15:13.736103 , only 7 left
Lucy got a b'caramel' at 2014-06-03 03:15:14.238152 , only 9 left
Lucy got a b'cherry' at 2014-06-03 03:15:14.739561 , only 8 left
```

可怜的露西！！

(5) 使用 ZeroMQ 发布第 7 章练习 (7) 中的诗，每次发布一个单词。写一个 ZeroMQ 客户端来打印出每个以元音开头的单词，再写另一个客户端来打印出所有长度为 5 的单词。忽略标点符号。

下面是服务器 poem_pub.py，把每个单词从诗中拆分出来。如果单词首字母为元音，就发布到主题 `vowels` 上；如果单词有五个字母，就发布到主题 `five` 上。一些词可能同时包括在两个主题内，也有一些都没有。

```
import string
import zmq

host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
pub.bind('tcp://%s:%s' % (host, port))

with open('mammoth.txt', 'rt') as poem:
    words = poem.read()
for word in words.split():
    word = word.strip(string.punctuation)
    data = word.encode('utf-8')
    if word.startswith(('a','e','i','o','u','A','e','i','o','u')):
        pub.send_multipart([b'vowels', data])
    if len(word) == 5:
        pub.send_multipart([b'five', data])
```

客户端程序 poem_sub.py，订阅主题 `vowels` 和 `five`，并打印输出主题和单词：

```
import string
import zmq

host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
sub = ctx.socket(zmq.SUB)
sub.connect('tcp://%s:%s' % (host, port))
sub.setsockopt(zmq.SUBSCRIBE, b'vowels')
sub.setsockopt(zmq.SUBSCRIBE, b'five')
while True:
    topic, word = sub.recv_multipart()
    print(topic, word)
```

如果你开启这些服务并执行代码，它们几乎是不工作的。代码看起来是对的，但是没有做任何事情。首先需要阅读 ZeroMQ 文档（http://zguide.zeromq.org/page:all）了解慢连接（slow joiner）问题：即使是在服务器端之前开启客户端，服务器会立刻发布数据，客户端只有片刻时间连接到服务器。如果你发布持续的数据流，当订阅的客户错过一些

是没有关系的。但在本例中，数据流很小导致订阅客户端“眨眼”间流过，就像快球迅速掠过击球手。

最简单的解决办法是在发布者（服务器端）调用 bind() 函数之后和开始发送消息之前休眠一秒。使用这个版本的程序 poem_pub_sleep.py：

```
import string
import zmq
from time import sleep

host = '127.0.0.1'
port = 6789
ctx = zmq.Context()
pub = ctx.socket(zmq.PUB)
pub.bind('tcp://%s:%s' % (host, port))

sleep(1)

with open('mammoth.txt', 'rt') as poem:
    words = poem.read()
for word in words.split():
    word = word.strip(string.punctuation)
    data = word.encode('utf-8')
    if word.startswith(('a','e','i','o','u','A','e','i','o','u')):
        print('vowels', data)
        pub.send_multipart([b'vowels', data])
    if len(word) == 5:
        print('five', data)
        pub.send_multipart([b'five', data])
```

开启订阅者进程，然后打开休眠版的发布者进程：

```
$ python poem_sub.py
```

```
$ python poem_pub_sleep.py
```

现在，订阅者有时间来捕捉这两个主题的消息。以下是它输出的前几行：

```
b'five' b'queen'
b'vowels' b'of'
b'five' b'Lying'
b'vowels' b'at'
b'vowels' b'ease'
b'vowels' b'evening'
b'five' b'flies'
b'five' b'seize'
b'vowels' b'All'
b'five' b'gaily'
b'five' b'great'
b'vowels' b'admired'
```

如果你不能在发布者程序中加入 sleep() 函数，可以使用 REQ 和 REPsockets 同步进行发布者和订阅者。在 GitHub（https://github.com/zeromq/pyzmq/tree/master/examples/pubsub）查看实例代码 publisher.py 和 subscriber.py。

附录 F

速查表

我发现我会频繁地查找某些东西。下面列出的表格希望对你有所帮助。

F.1 操作符优先级

下面这张表是官方文档中关于优先级的混合，高优先级的运算符在上面。

操作符	描述和示例
[v1, ...]、{v1, ...}、{k1: v1, ...}、(...)	列表 / 集合 / 字典 / 生成器的创建和推导，括号内表达式
seq[n]、seq[n:m]、func(args...)、obj .attr	索引、切片、函数调用和属性引用
**	幂运算
\`+\`x、\`-\`x、\`~\`x	正号、负号和位求反
*、/、//、%	乘法、浮点除法、整数除法和取余
+、-	加法、减法
<<、>>	按位左移、按位右移
&	按位与
|	按位或
in、not in、is、is not、<、<=、>、>=、!=、==	属于关系和相等性测试
not *x*	布尔取非
and	布尔取与
or	布尔取或
if...else	条件表达式
lambda	lambda 表达式

F.2 字符串方法

Python 不仅提供了字符串方法（不借助 str 对象），而且包含了定义丰富的 string 模块。使用下面的测试变量：

```
>>> s = "OH, my paws and whiskers!"
>>> t = "I'm late!"
```

F.2.1 改变大小写

```
>>> s.capitalize()
'Oh, my paws and whiskers!'
>>> s.lower()
'oh, my paws and whiskers!'
>>> s.swapcase()
'oh, MY PAWS AND WHISKERS!'
>>> s.title()
'Oh, My Paws And Whiskers!'
>>> s.upper()
'OH, MY PAWS AND WHISKERS!'
```

F.2.2 搜索

```
>>> s.count('w')
2
>>> s.find('w')
9
>>> s.index('w')
9
>>> s.rfind('w')
16
>>> s.rindex('w')
16
>>> s.startswith('OH')
True
```

F.2.3 修改

```
>>> ''.join(s)
'OH, my paws and whiskers!'
>>> ' '.join(s)
'O H ,   m y   p a w s   a n d   w h i s k e r s !'
>>> ' '.join((s, t))
"OH, my paws and whiskers! I'm late!"
>>> s.lstrip('HO')
', my paws and whiskers!'
>>> s.replace('H', 'MG')
'OMG, my paws and whiskers!'
>>> s.rsplit()
['OH,', 'my', 'paws', 'and', 'whiskers!']
>>> s.rsplit(' ', 1)
['OH, my paws and', 'whiskers!']
>>> s.split()
['OH,', 'my', 'paws', 'and', 'whiskers!']
>>> s.split(' ')
['OH,', 'my', 'paws', 'and', 'whiskers!']
>>> s.splitlines()
['OH, my paws and whiskers!']
```

```
>>> s.strip()
'OH, my paws and whiskers!'
>>> s.strip('s!')
'OH, my paws and whisker'
```

F.2.4　格式化

```
>>> s.center(30)
'  OH, my paws and whiskers!   '
>>> s.expandtabs()
'OH, my paws and whiskers!'
>>> s.ljust(30)
'OH, my paws and whiskers!     '
>>> s.rjust(30)
'     OH, my paws and whiskers!'
```

F.2.5　字符串类型

```
>>> s.isalnum()
False
>>> s.isalpha()
False
>>> s.isprintable()
True
>>> s.istitle()
False
>>> s.isupper()
False
>>> s.isdecimal()
False
>>> s.isnumeric()
False
```

F.3　字符串模块属性

这些是用于常量定义的类属性：

<table>
<tr><th>属性</th><th>示例</th></tr>
<tr><td>ascii_letters</td><td>'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'</td></tr>
<tr><td>ascii_lowercase</td><td>'abcdefghijklmnopqrstuvwxyz'</td></tr>
<tr><td>ascii_uppercase</td><td>'ABCDEFGHIJKLMNOPQRSTUVWXYZ'</td></tr>
<tr><td>digits</td><td>'0123456789'</td></tr>
<tr><td>hexdigits</td><td>'0123456789abcdefABCDEF'</td></tr>
<tr><td>octdigits</td><td>'01234567'</td></tr>
<tr><td>punctuation</td><td>"!"#$%&\'()*+,-./:;<=>?@[\]^_{|}~’</td></tr>
<tr><td>printable</td><td>"0123456789abcdefghijklmnopqrstuvwxyz' + 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' +
'!"#$%&\'()*+,-./:;<=>?@[\]^_`{|}~’ + ' \t\n\r\x0b\x0c'</td></tr>
<tr><td>whitespace</td><td>' \t\n\r\x0b\x0c'</td></tr>
</table>

作者介绍

Bill Lubanovic 1977 年开始开发 Unix 软件，1981 年开始开发 GUI 软件，1990 年开始开发数据库软件，1993 年开始开发 Web 软件。

1982 年，Bill 在一家名为 Intran 的创业公司为最早的图形工作站之一开发了 MetaForm，这是（在 Mac 和 Windows 之前）最早的商业 GUI 软件之一。20 世纪 90 年代初期，Bill 在 Northwest Airlines 编写了一个图形化的收益管理系统，它带来了数百万美元的收入；Bill 还帮助公司在互联网上打响了名声，并编写了公司的第一个网络营销测试。之后，他在 1994 年和别人共同成立了 ISP 公司（Tela），在 1999 年成立了一家 Web 开发公司（Mad Scheme）。

近几年，他还和一个远程团队一起为曼哈顿的一家创业公司编写核心服务。目前，他正在为一家超级计算机公司集成 OpenStack 服务。

Bill 很享受在明尼苏达州的生活，陪伴他的还有他优秀的妻子 Mary、孩子 Tom 和 Karin 以及猫咪 Inga、Chester 和 Lucy。

封面介绍

本书封面上的动物是一只亚洲巨蟒（网纹蟒）。这种蛇并不像看起来那样吓人：它没有毒性，也很少攻击人类。这种蛇的长度可以达到七米（有时甚至能超过九米），是世界上最长的蛇和爬行动物，不过大多数个体只有三四米长。在拉丁语中，这种蛇的名字形容的是它像网一样的图案和颜色。不同地区网纹蟒的尺寸和颜色差别很大，但是背部都有钻石形状。特殊的外表可以让网纹蟒轻松地融入周围的环境中。

亚洲巨蟒主要分布在东南亚。按照地区可以划分为三个亚种，但是科学界并不认可这种说法。蟒蛇通常生活在雨林、树林、草原和水中，它们的游泳技能十分出众，甚至能游到很远的岛上。蟒蛇主要的食物是哺乳动物和鸟类。

网纹蟒在人工饲养中越来越受欢迎，因为它们的外表很有特点，习性良好。不过在饲养中也会遇到一些问题。有记录表明出现过亚洲巨蟒吃人或者杀人的案例。对于活体蟒蛇的长度测量也非常困难，只能准确测量已经死亡或者麻醉的蟒蛇。人工饲养的蟒蛇大多比野生蛇更胖一些，而且它们通常来说不具有危险性，顶多算是不太稳定。

O’Reilly 封面上出现的许多动物都濒临灭绝，但是它们对地球来说都非常重要。如果想伸出援手，请访问 http://animals.oreilly.com。

封面图片来自 Johnson 的《自然历史》。

版权声明

欢迎加入

图灵社区 iTuring.cn

——最前沿的IT类电子书发售平台

电子出版的时代已经来临。在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。作为国内第一家发售电子图书的IT类出版商，图灵社区目前为读者提供两种DRM-free的阅读体验：在线阅读和PDF。

相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子书出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

优惠提示：现在购买电子书，读者将获赠书款20%的社区银子，可用于兑换纸质样书。

——最方便的开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版和开源出版的梦想。利用“合集”功能，你就能联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者。（收费形式须经过图灵社区立项评审。）这极大地降低了出版的门槛。只要你有写作的意愿，图灵社区就能帮助你实现这个梦想。成熟的书稿，有机会入选出版计划，同时出版纸质书。

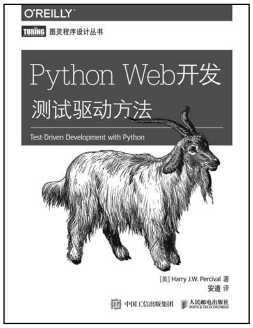
图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果你有意翻译哪本图书，欢迎你来社区申请。只要你通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

——最直接的读者交流平台

在图灵社区，你可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。

你可以积极参与社区经常开展的访谈、乐译、评选等多种活动，赢取积分和银子，积累个人声望。

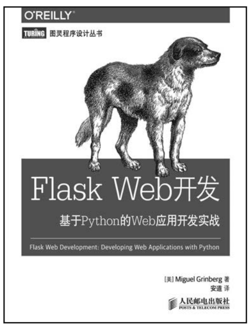
延 展 阅 读



实战式 TDD 开发指南，使用 Python、Django 等流行技术开发现代 Web 应用！

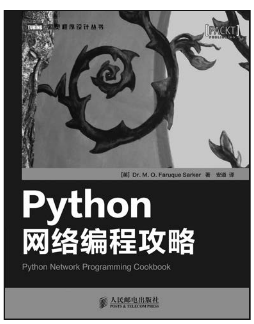
本书手把手教你从头开始开发一个真正的Web应用，并且展示使用Python做测试驱动开发（TDD）的优势。你将学到如何在开发应用的每一个部分之前先编写和运行测试，然后再编写最少量的代码让测试通过。也就是说，你将学会应用TDD理念，写出简洁可用、赏心悦目的代码。

书号：978-7-115-40327-8
定价：99.00 元



作者集 25 年软件开发经验，展示自己使用 Flask 开发 Web 程序的工作流程
从安装与环境设置讲起，一步一步搭建服务器端 Web 应用
一个程序贯穿始终，从寥寥几行代码逐章扩展，最终打造成功能完善的社交博客 Flasky
直接给出必知知识，为初学者提供进一步探索的起点
全流程讲解 Web 应用开发，让中高级读者掌握最佳实践

书号：978-7-115-37399-1
定价：59.00 元



Python是门很棒的编程语言，适合快速构建应用原型。本书全面介绍了Python网络编程涉及的重要问题，包括网络编程、系统和网络管理、网络监控以及Web应用开发。作者通过70多篇攻略，清晰简明地描述了各种网络任务和问题，提出了可用于多种场景的解决方案，并细致地分析了整个操作过程。如果你想开发依赖于网络协议的实用Web应用和网络应用，绝对不能错过这本书。

书号：978-7-115-37269-7
定价：45.00 元



程序员爸爸的第一本亲子互动编程书！

编程是一项充满乐趣的挑战，想要上手也非常容易！本书中，Warren和Carter父子以亲切的笔调、通俗的语言，透彻全面地介绍了计算机编程世界。他们以简单易学的Python语言为例，通过可爱的漫画、有趣的例子，生动地介绍了变量、循环、输入和输出、数据结构以及图形用户界面等编程的基本概念。只要懂得计算机的基本操作，如启动程序、保存文件，任何人都可以跟随本书，由简入难，学会编写程序，甚至制作游戏。本书内容经过教育专家的评审，经过孩子的亲身检验，并得到了家长的认可。

书号：978-7-115-36717-4
定价：69.00 元

O'REILLY®

Python语言及其应用

本书内容易于理解，而且读起来生动有趣，是编程和Python初学者不可多得的优秀教程。书中首先介绍了Python的基础知识，然后逐渐深入多种主题，结合教程和菜谱式风格来讲解Python 3中的概念。每章结尾的练习可以帮助你巩固所学的知识。

本书会为你学习Python打下坚实的基础，包括测试、调试、代码复用的最佳实践以及其他开发技巧。同时还会告诉你如何在商业、科学和艺术领域使用Python，并教会你使用多种Python工具和开源包。

通过阅读本书，你将能够：

- **学习简单的数据类型，以及基本的数学和文本操作**
- **学习用Python内置的数据结构来处理数据**
- **掌握Python的代码结构和函数的用法**
- **使用模块和包编写大规模Python程序**
- **深入理解对象、类和其他面向对象特性**
- **学习使用普通文件、关系数据库和NoSQL数据库来存储数据**
- **使用Python构建Web客户端、服务器、API和服务**
- **管理系统任务，比如程序、进程和线程**
- **理解并发和网络编程的基础知识**

Bill Lubanovic现为Penguin Computing公司高级软件工程师。1977年开始开发Unix软件，1981年开始开发GUI软件，1990年开始开发数据库软件，1993年开始开发Web软件。与人合著有*Linux System Administration*。

"Bill Lubanovic编写了一本非常优秀的著作，介绍了编程的基础知识，并指导你如何利用庞大的Python工具箱解决现实生活中的问题。通过阅读本书，你一定能学会如何用Python解决问题。"

——**Loïc Pefferkorn**
开源系统工程师

PYTHON

封面设计：Ellie Volckhausen　张健

图灵社区：iTuring.cn
热线：(010)51095186转600

分类建议 计算机／程序设计／Python

人民邮电出版社网址：www.ptpress.com.cn

O'Reilly Media, Inc.授权人民邮电出版社出版

ISBN 978-7-115-40709-2

9 787115 407092

ISBN 978-7-115-40709-2
定价：79.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks